

MATRIX FACTORIZATION USING A
BLOCK-RECURSIVE STRUCTURE AND
BLOCK-RECURSIVE ALGORITHMS

Jeremy D. Frens

Submitted to the faculty of the Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University

August 2002

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Doctoral Committee

David S. Wise
(Principal Advisor)

Randall Bramley

Dennis Gannon

May 22, 2002

Daniel Maki

Acknowledgements

This work was supported, in part, by the National Science Foundation under grant number CDA93-03189 and by the U.S. Department of Education under grant numbered P200A50237.

I first thank the members of my committee for helping me through this dissertation. Thanks especially to my advisor David S. Wise for all of his help and encouragement. Thanks to the other members of the committee, Randy Bramley, Dennis Gannon, and Daniel Maki, who had to stretch their memories to recall who I was when I started scheduling my defense. A special thanks to Randy who reminded me exactly what I needed to do.

For technical assistance, my thanks to Bruce Shei who helped me many times when I had problems with and questions about the various machines. Without Bruce, this dissertation wouldn't be here. Also, thanks to Greg Alexander who wrote a library for parallel dispatch system for quadtree matrices. I'm not sure what I would have done without that library.

Thanks also to all of my colleagues at both Northwestern College and Calvin College. Thanks for the support and encouragement that everyone from both institutions gave me.

My friends in Bloomington, Orange City, and Grand Rapids also deserve a lot of thanks. Those I knew in Bloomington: Dave Wilson for agreeing to room with me for three years; Kyle and Amy Wagner for their friendship and spaghetti nights; Peter Weingartner, Eric Hilsdale, and Susan Lato for their friendships and tolerance of lengthy discussions, usually about nothing; and all of my other friends at IU who aren't listed here. Those I knew in Orange City: Jeff and Jana Boersma who were familiar faces from a previous era; Mark Vellinga who was an excellent colleague; and all of the students I knew at Northwestern. And now those I knew and know in Grand Rapids, especially to Marie Albers who still remains a best friend after so many years. Also thanks to my Computer Science colleagues at Calvin; a special thanks to Joel Adams and David Laverell for proofreading my dissertation at the last minute.

Finally, my greatest thanks goes to my family. First to my family at United Presbyterian Church in Bloomington. You were my family for six years while I lived there, and I owe you more than I could ever repay. A special thanks to David Bremer who was a good cross between a brother and a father for me. Most importantly, I thank my biological family. Thanks to my Uncle Bud and Aunt Ruth who look after me now just as they did earlier. Thanks to my grandparents who have been supportive throughout my education. Thanks to sister Staci, brother Joel, and sister-in-law Rachel who befriended me while we were all at Calvin. And thanks to my parents who have done more for me than should be required of any parents; it was because of your encouragement and examples that got me here. This dissertation is for you, Mom and Dad.

Abstract

The divide-and-conquer paradigm yields algorithms that parallelize easily, a very important consideration in high-performance computing. However, high-performance computing also relies on local reuse of data in a memory hierarchy of registers, caches, main memory, and swapping disks.

I have worked with a sequential representation of quadtree matrices that is a divide-and-conquer data structure. This representation uses an indexing scheme, Morton ordering, that automatically blocks the elements of the matrix and promotes memory locality in recursive algorithms over the quadtree. This work focuses on using this representation for two important matrix algorithms, matrix multiplication and QR factorization.

Techniques for generating independent and local code were discovered while implementing a recursive matrix-matrix multiplication over the sequential quadtree matrix. For good memory locality the basic multiplication routine was written as two dual, mutually recursive functions with the recursive calls in each version precisely ordered to reuse data already in cache.

To avoid unnecessary work, minimal decorations annotate each submatrix of the matrix; these decorations are used to ignore zero blocks and to elide the zero tests on

blocks known to be dense. Finally, a strategy was developed for dispatching parallel processes in the multiplication algorithm.

These lessons were then used to develop an efficient QR factorization algorithm for quadtree matrices. It also uses recursive functions that localize data in blocks and that balance parallel processing.

The sequential quadtree matrix and multiplication functions are implemented in C because of its optimizing compilers. Since C optimizers favor iterative code and are deficient for recursive code, some base optimizations were done by hand. These hand optimizations could and should be done by some future optimizing compiler.

Results are mostly favorable; while the quadtree-matrix algorithms do not always perform at the same level as decades-old routines (i.e., BLAS and LAPACK), which are fine-tuned for traditional storage, the quadtree algorithms are competitive and even expose some flaws in these old routines.

Contents

Acknowledgements	iii
Abstract	v
1 Introduction	1
1.1 Divide-and-Conquer Paradigm for High-Performance Computing	1
1.2 Parallelism	2
1.2.1 Divide-and-Conquer Algorithms	2
1.2.2 Divide-and-Conquer Parallelism	4
1.3 Efficient Memory Use	5
1.3.1 Divide-and-Conquer Data Structures	5
1.3.2 The Memory Hierarchy	6
1.4 Problems in High-Performance Computing	9
1.4.1 Matrix Terminology	9

1.4.2	Matrix Multiplication	12
1.4.3	Solving a Linear System	12
1.4.4	Problems with Matrix Algorithms	13
1.5	Traditional Matrix Storage	14
1.6	The Quadtree Matrix	15
1.6.1	Trees	15
1.6.2	Quadtree Indexings	17
1.6.3	Quadtree Matrix	19
1.6.4	Quadtree Matrix Implementation	22
1.6.5	Decorating Quadtree Matrices	24
1.6.6	Storage Arrays	25
1.6.7	A Divide-and-Conquer Data Structure	27
1.7	Row Major Versus Morton Order	28
1.8	Related Research	29
1.9	New Contributions	31
1.10	Road Map	32
2	High-Performance Environment	33
2.1	Proof of Concept	33
2.2	Programming Liberties	34

2.3	Machines	35
2.4	Compiler Issues	35
2.4.1	Compilers and Libraries	35
2.4.2	Stride	37
2.4.3	Recursive Code	37
2.4.4	Compiler Flags	38
2.4.5	Shared Memory and Parallel Dispatch	38
2.5	Timing Issues	40
2.5.1	Measuring Performance	40
2.5.2	Machine Solutions	41
3	Matrix Multiplication	42
3.1	Row-Major Multiplication	42
3.2	Quadtree Matrices	44
3.2.1	Two Versions of Matrix Multiplication	44
3.2.2	Analytical Proof of Good Memory Behavior	46
3.3	Successful Matrix Multiplication	49
3.4	Compiler and Implementation Issues	50
3.4.1	BLAS	50
3.4.2	Unfolding the Base Case	50

3.4.3	Decoration Driven Multiplication	52
3.4.4	Destructive Multiplication	53
3.4.5	Decorating C	55
3.4.6	Top Level Function	56
3.5	Experimental Results	56
3.5.1	Base Case Size	57
3.5.2	Flop Counts	58
3.5.3	Quadtree Algorithm Versus BLAS Algorithm	58
4	Parallel Matrix Multiplication	65
4.1	Dispatching Parallel Processes	65
4.2	Eight Multiplication Patterns	66
4.3	Implementation and Compiler Issues	75
4.3.1	Padding Size	76
4.3.2	Uniprocessor-Commit Size	76
4.4	Experimental Results	77
5	QR Factorization	81
5.1	The Basic Algorithms	81
5.1.1	Householder QR Factorization	82
5.1.2	Givens QR Factorization	82

5.2	The Quadtree QR Factorization Algorithms	84
5.2.1	Quadtree QR Factorize	84
5.2.2	Quadtree QR Eliminate	84
5.3	Tuning for the Memory Hierarchy	86
5.4	Successful QR Factorization	88
5.4.1	Computational Correctness	88
5.4.2	Avoiding Problematic Operations	89
5.4.3	Error Analysis	90
5.5	Implementation and Compiler Issues	90
5.5.1	In-place Algorithms	90
5.5.2	The Transpose of Q	91
5.5.3	Base Case Unfolding	91
5.5.4	Decoration Driven QR Factorization	91
5.5.5	Accumulating Q	92
5.6	LAPACK	92
5.7	Experimental Results	93
5.7.1	Flop Count	93
5.7.2	Quadtree Algorithm Versus LAPACK Algorithm	94

6	Parallel QR Factorization	102
6.1	QR Parallel Cases	102
6.1.1	Parallel Patterns of f	103
6.1.2	Parallel Patterns of e	104
6.2	Parallel LAPACK	110
6.3	Experimental Results	110
7	Conclusion	115
7.1	Results	115
7.2	Comments	116
7.3	Future Work	117
7.3.1	Rectangular Matrices	117
7.3.2	Processor Dispatch	118
7.3.3	Concise Representation of Q	119
7.3.4	Sparse Matrices	119
7.3.5	Other Algorithms	120
7.3.6	Patterns in Algorithms	121
7.4	Curious Observations	121
7.5	Conclusion	122

A Proofs	125
A.1 Transpose Properties	125
A.2 Lemmas for Orthogonal Matrices	126
A.3 Correctness of QR Factorization	129
A.4 QR Factorization Success	142
References	144

List of Tables

1.1	Matrix notation	10
1.2	Correlation of quadrants to index computations	22
2.1	System parameters of Power Challenge	35
2.2	System parameters of Octane	36
2.3	System parameters of Enterprise 450	36
2.4	System parameters of Ultra 5/10	37
2.5	Compiler optimizations for SGIs	39
2.6	Compiler optimizations for Suns	39

List of Figures

1.1	Factorial function (Version A)	3
1.2	Factorial function (Version B)	3
1.3	Quicksort function	3
1.4	Row-major indexing	15
1.5	Column-major indexing	15
1.6	Level-order indexing of quaternary tree	17
1.7	Morton-order indexing of quaternary tree	18
1.8	Matrix M_1	19
1.9	Matrix M_1 padded	19
1.10	Quadtree matrix representation of $\text{pad}(M_1)$	19
1.11	Matrix M_2	20
1.12	Matrix M_2 padded	20
1.13	Quadtree matrix representation of $\text{pad}(M_2) \downarrow \mathbf{ne}$	20
1.14	Morton ordering of a 16×16 matrix with Z order	23

1.15	Level-order storage of $\text{pad}(M_1)$ decoration (cf. Figure 1.9)	26
1.16	Morton-order storage of $\text{pad}(M_1)$ scalars	26
1.17	Level-order storage of $\text{pad}(M_2 \downarrow \text{ne})$ decoration (cf. Figure 1.12)	27
1.18	Morton-order storage of $\text{pad}(M_2 \downarrow \text{ne})$ scalars	27
3.1	Nested loops for inner-product multiplication	42
3.2	Blocked nested loops for multiplication	43
3.3	Cache-poor quadtree matrix multiplication	45
3.4	Cache-friendly quadtree multiplication	46
3.5	Original unfolded 2×2 base case	51
3.6	Rolled 2×2 base case	51
3.7	Raised base case version of multiplication	53
3.8	First-visit version of multiplication	54
3.9	Last-visit version of multiplication	55
3.10	Top-level multiplication function	56
3.11	Running time of different base case sizes	57
3.12	Running time of uniprocessor multiplication on Power Challenge	61
3.13	Mflop/s of uniprocessor multiplication on Power Challenge	61
3.14	Running time of uniprocessor multiplication on Octane	62
3.15	Mflop/s of uniprocessor multiplication on Octane	62

3.16	Running time of uniprocessor multiplication on Enterprise 450	63
3.17	Mflop/s of uniprocessor multiplication on Enterprise 450	63
3.18	Running time of uniprocessor multiplication on Ultra 5/10	64
3.19	Mflop/s of uniprocessor multiplication on Ultra 5/10	64
4.1	Parallel multiplication patterns	67
4.2	Parallel dispatch chart	67
4.3	Parallel multiplication: perfect square dispatch	70
4.4	Parallel multiplication: perfect square dispatch (4 processes)	70
4.5	Parallel multiplication: square•square dispatch	70
4.6	Parallel multiplication: square•colonnade majority-padding dispatch .	72
4.7	Parallel multiplication: square•colonnade minority-padding dispatch .	72
4.8	Parallel multiplication: stripe•square majority-padding dispatch . . .	72
4.9	Parallel multiplication: stripe•square minority-padding dispatch . . .	72
4.10	Parallel multiplication: colonnade•square majority-padding dispatch .	74
4.11	Parallel multiplication: colonnade•square minority-padding dispatch .	74
4.12	Parallel multiplication: square•stripe majority-padding dispatch . . .	74
4.13	Parallel multiplication: square•stripe minority-padding dispatch . . .	74
4.14	Parallel multiplication: colonnade•stripe majority-padding dispatch .	74
4.15	Parallel multiplication: colonnade•stripe minority-padding dispatch .	74

4.16	Parallel multiplication: stripe•colonnade majority-padding dispatch	75
4.17	Parallel multiplication: stripe•colonnade minority-padding dispatch	75
4.18	Speed-up for quadtree multiplication on Power Challenge	79
4.19	Speed-up for <code>dgemm()</code> on Power Challenge	79
4.20	Speed-up for quadtree multiplication on Enterprise 450	80
4.21	Speed-up for <code>dgemm()</code> on Enterprise 450	80
5.1	Iterative, column-based QR factorization using Givens rotations	83
5.2	QR factorization function, f	85
5.3	QR elimination function, e	87
5.4	Flop count of QR factorization algorithms	93
5.5	Running time of uniprocessor QR factorization on Power Challenge	98
5.6	Mflop/s of uniprocessor QR factorization on Power Challenge	98
5.7	Running time of uniprocessor QR factorization on Octane	99
5.8	Mflop/s of uniprocessor QR factorization on Octane	99
5.9	Running time of uniprocessor QR factorization on Enterprise 450	100
5.10	Mflop/s of uniprocessor QR factorization on Enterprise 450	100
5.11	Running time of uniprocessor QR factorization on Ultra 5/10	101
5.12	Mflop/s of uniprocessor QR factorization on Ultra 5/10	101
6.1	Parallel patterns of f	103

6.2	Parallel QR factorization: perfect square dispatch	105
6.3	Parallel QR factorization: square and stripe dispatch	105
6.4	Parallel patterns of e	106
6.5	Parallel QR elimination: perfect square dispatch	107
6.6	Parallel QR elimination: stripe majority-padding dispatch	108
6.7	Parallel QR elimination: stripe minority-padding dispatch	109
6.8	Speed-up for quadtree QR factorization on Power Challenge	113
6.9	Speed-up for quadtree QR factorization on Enterprise 450	114
6.10	Speed-up for <code>dgeqrf()</code> on Enterprise 450	114
7.1	Modified parallel QR elimination (perfect square) dispatch	122

Introduction

1.1 Divide-and-Conquer Paradigm for High-Performance Computing

This dissertation addresses the divide-and-conquer paradigm in the context of high-performance computing, where data sets are huge and algorithms take a long time to run. High-performance computing has two main concerns for efficient programs. The first concern is parallelism [50] which has long been a promise of the divide-and-conquer paradigm: compute all independent expressions in parallel [13, Section 1.3.1][3]. The second concern of high-performance computing is the reuse of a computer's memory [37, Chapter 7], respecting the different types of memory in a computer and programming accordingly. This concern also has a divide-and-conquer solution.

Divide-and-conquer is a common tool in algorithm design, but crucial to functional programming [8] which motivated this research. Functional programming is a

paradigm where computations are expressed using only functions and arguments, excluding side-effects and hence unnecessary sequentiality; so syntactically independent subexpressions are computationally independent as well. The divide-and-conquer paradigm splits a problem into a few *independent* and *local* subproblems.

1.2 Parallelism

The divide-and-conquer paradigm has long promised and delivered good parallelism. A couple of classic examples demonstrate how.

1.2.1 Divide-and-Conquer Algorithms

A divide-and-conquer algorithm divides a problem into multiple subproblems, and often the subproblems can be solved independently of each other. Solutions for the subproblems are later combined together to yield a solution for the original problem.

The canonical example of functional programming is the factorial function as written in Figure 1.1 [13, p. 6] using Haskell [8]. However, this formulation does not use divide-and-conquer since there is only one function call (i.e., one subproblem) in the recursive case. Divide-and-conquer is used in the second version of the factorial function in Figure 1.2 [13, p. 6] through the use of a second function `prod`. The function `prod` multiplies a sequence of consecutive integers by splitting the sequence in two and calling itself recursively on each half.

The divide-and-conquer paradigm can also be used for sorting elements in a list. Quicksort [27] is one such algorithm, presented in Figure 1.3 [38, p. 119] also using

```
facA 0 = 1
facA n = n * facA (n-1)
```

Figure 1.1: Factorial function (Version A)

```
facB 0 = 1
facB n = prod 1 n
prod n m = if m=n then m
          else (prod m halfway) * (prod halfway+1 n)
          where halfway = m + ((n-m) div 2)
```

Figure 1.2: Factorial function (Version B)

```
qsort [] = []
qsort (pivot:rest) = qsort lower ++ [pivot] ++ qsort upper
  where lower = [ x | x <- rest, x <= pivot]
        upper = [ x | x <- rest, x > pivot]
```

Figure 1.3: Quicksort function

Haskell. The inductive case for the `qsort` function receives a list as indicated by the colon operator¹: the first element of the list is called `pivot`, and the rest of the list is aptly named `rest`. The rest of the list is partitioned into two separate lists, `upper` and `lower`, by “pivoting” around `pivot` using two list comprehensions². The list `lower` will contain elements less than or equal to the pivot; the list `upper` will contain elements greater than the pivot. Both of these lists are sorted recursively and then concatenated into a result with the infix list-concatenation operator `++`.

¹The colon operator is the infix `cons` operator; Haskell allows the operator to be used in the pattern matching of parameters. [8, Section 4.1.1]

²Read list comprehensions as set notation with the vertical bar `|` meaning “such that” and the left arrow operator `<-` meaning “element of”. [8, Section 4.3.2]

1.2.2 Divide-and-Conquer Parallelism

An essential step in writing parallel algorithms is partitioning computations into *processes* that can be executed in parallel. The divide-and-conquer paradigm offers a straightforward way of writing parallel code: take the divide-and-conquer algorithm and solve independent subproblems in parallel.

For example, the two recursive calls in `prod` in Figure 1.2 might be done in parallel. The quicksort algorithm of Figure 1.3 has two opportunities for parallelism: (1) computing `upper` and `lower`, and (2) the two recursive calls to `qsort`.

Scheduling the processes is complicated by several factors. One factor is inter-process communication [1, 14, 28] which must be minimized. When the divide-and-conquer paradigm schedules independent computations in parallel, there is no inter-process communication while these processes execute. Inter-process communication is needed only as parallel processes are dispatched and after they end. Consequently, the number of dispatches should be minimized, and this is accomplished by dispatching parallel processes early in the recursive decomposition.

Even more importantly, all of the available processors must be kept busy with useful work. One simple solution is to dispatch parallel processes that are given the same amount of work as each other; each process is given an equal number of processors for further dispatch. The function `prod` (Figure 1.2) balances its parallel processes quite well: each process is given half of the sequence to multiply together. Each process would be given half of the available processors for further dispatch.

Some algorithms are not as elegant. For example, balancing the parallel processes of quicksort (Figure 1.3) cannot use the same elegant solution as `prod`. The two list

comprehensions can be done in parallel; both list comprehensions take about the same amount of work (i.e., both process the whole original list) resulting in well balanced processes. However, depending on the pivot element, the lengths of `upper` and `lower` may be quite different from each other, resulting in significantly different amounts of work to sort each one individually. For these parallel processes, each process could be given a number of processors proportional to the length of the list given to that process.

1.3 Efficient Memory Use

The divide-and-conquer paradigm has long been used to design a variety of abstract data structures. However, often in functional programming, the implementations of these data structures have been done using linked structures. Linked structures tend to suffer in performance in high-performance computing because of data locality problems, so other solutions must be used. The divide-and-conquer paradigm offers one solution.

1.3.1 Divide-and-Conquer Data Structures

Divide-and-conquer can be applied in at least two ways for a data structure.

First, the divide-and-conquer paradigm can be used in the design of the abstract data structure. Trees are the most common divide-and-conquer data structures. (See

Section 1.6.1 for a more formal discussion of trees.) Binary search trees [30, Section 6.2.2] and multi-way trees like B-trees [30, Section 6.2.4] are dictionary structures that partition the data, one partition for each subtree. The partitioning is done deliberately so that common operations like insert and search are very efficient, proportional to the height of the tree.

Divide-and-conquer algorithms arise naturally from exploiting the substructures of a divide-and-conquer data structure. Tree traversals [29, Section 2.3.1] can be divide-and-conquer algorithms.

Second, the divide-and-conquer paradigm can also be applied to the manifestation of a data structure; that is, it can be used to determine how to map the abstract data structure directly into a computer’s memory. Row-major storage of a matrix (see Section 1.5) follows this approach: divide the matrix into rows, and map each element of each row into consecutive locations in memory. The sequential storage of the heap used in heapsort [30, p. 144] also follows this approach: store the levels of the heap (a tree) in consecutive location in memory.

1.3.2 The Memory Hierarchy

The memory of a modern computer is layered in a hierarchy, top to bottom³: primary cache, secondary cache, main memory, virtual memory, and distributed memory—with more levels to come in the future.

Definition 1.1 *The transfer block of a level of the memory hierarchy is the smallest*

³This dissertation follows the tradition that memories closer to the processor are higher in the hierarchy (see Patterson and Hennessy [37, p. 542]); this is by no means universal (see Whaley and Dongarra [45]).

block of contiguous memory uploaded to that level of the memory hierarchy.

This is a generalization of familiar concepts: a cache line is the transfer block of a cache; a page of virtual memory is the transfer block of a virtual memory system into main memory. Patterson and Hennessy [37, p. 542] use the term “block” for this same concept.

Definition 1.2 *A memory miss is a memory access that triggers the transfer of a transfer block into a level of the memory hierarchy.*

This, too, is a generalization of familiar concepts: a cache miss is a memory miss in a cache; a page fault is a memory miss in virtual memory. Again, see Patterson and Hennessy [37, p. 542].

The different types of memory in the hierarchy have different relative speeds and sizes. The top of the hierarchy is very fast; the bottom of the hierarchy is very slow. The size of transfer blocks tend to be small at the top and large on the bottom. The memories themselves are similarly small at the top and large on the bottom.

The large problems of high-performance computing and the cost of faster memories force the use of the slower memories lower in the hierarchy, and efficiency demands that algorithms minimize the number of times that the slower memories are accessed. As memory speeds fall further and further behind processor speeds [49], this demand becomes more and more important.

The number of memory misses can be reduced by increasing data locality. The memory hierarchy is managed with two types of locality in mind, temporal and spatial. *Temporal locality* suggests that once a data item is used, it will be used again soon.

Spatial locality suggests that if a program accesses one memory location, it will also access nearby memory locations. (See Patterson and Hennesy [37, p. 540].) Spatial locality explains why transfer blocks are blocks of contiguous memory. Temporal locality in turn suggests a programming style: when a memory location is accessed, use it and nearby data as much as possible, where “nearby” is defined by the transfer blocks of each level of the memory hierarchy.

Traditional analysis of algorithms ignores memory accesses by focusing on computational operations and assuming all accesses are done in constant time. Due to the variety of access times at different levels of the memory hierarchy, this assumption results in a misleading analysis. A more realistic analysis would account for the inevitable memory misses and the amount of time needed to access different memories. Asymptotically, this careful analysis is the same as the naive analysis; the difference is only apparent on the coefficients of the formulas for the analyses. Yet, often these coefficients are extremely significant. (After all, parallel speed-up is usually just a modification of the coefficient of the analysis.)

Programming with traditional storage requires an expert programmer or an optimizing compiler to respect the transfer blocks of the memory hierarchy. Often the sizes of these transfer blocks are used by this expert programmer, making the code unportable. This knowledge can be non-trivial since different computers have different types of memories with different sizes. It is better to have the compiler deal with this knowledge since it should be relatively easy for it to collect. Furthermore, since each type of computer can be configured with differently sized memories (especially RAM and virtual memory), the code should be compiled for each physical machine.

The divide-and-conquer paradigm offers a solution to all of these problems. Too often functional programming has ignored data locality, favoring linked structures whose components are scattered throughout memory. This oversight causes problems for computation-intensive algorithms since the links may routinely cross transfer blocks causing many memory misses. But the divide-and-conquer paradigm applied to sequential storage offers a solution: map a data structure into memory by allocating blocks of contiguous memory for the substructures of a data structure. Memory can be allocated this way recursively down to the base case of the data structure so that every substructure at every level of the data structure is mapped into contiguous memory. A program then takes advantage of data locality simply by manipulating the substructures without knowledge of the machine's parameters.

1.4 Problems in High-Performance Computing

Typical problems in high-performance computing involve matrices. These problems are prime candidates for demonstrating how effective the divide-and-conquer paradigm can be for high-performance computing.

1.4.1 Matrix Terminology

Informally, a matrix is a two-dimensional grid of real numbers represented in the computer as floating point values [24, Section 2.4]. For example, if A is an $m \times n$

Symbol	Definition
a_{ij}	Element of matrix A in row i , column j
$A_{\alpha\beta}$	Block in stripe α , colonnade β in matrix A (see Definition 1.9)
A^T	the transpose of A
A^{-1}	the inverse of A
I	the identity matrix
Z	the zero matrix
$A \downarrow dd$	quadrant dd of matrix A where dd can be nw , ne , sw , or se .

Standard matrix notation taken from Golub and Van Loan [24].

Table 1.1: Matrix notation

matrix (i.e., m rows and n columns), its grid would look like this:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

See Table 1.1 for the notation used for matrices.

Several basic terms are useful when talking about matrices:

Definition 1.3 *The transpose A^T of an $m \times n$ matrix A is the $n \times m$ matrix obtained from A by interchanging the rows with the columns; that is, $a_{ij}^T = a_{ji}$. [21, p. 15]*

One of the most common matrix operations is matrix multiplication:

Definition 1.4 *Matrix-matrix multiplication is written $C = AB$, where A is an*

$m \times p$ matrix, B is an $p \times n$ matrix, and C is an $m \times n$ matrix; it is defined by

$$c_{ij} = \sum_{k=1}^p a_{ik}b_{kj}, \quad (1.1)$$

for all $1 \leq i \leq m$ and $1 \leq j \leq n$. [24, Section 1.1.2]

Definition 1.5 *The elements of the identity matrix I are all zero except for the elements on the main diagonal (where the row and column indices are equal) which are set to 1. (See Friedberg et al. [21, p. 75].)*

The identity matrix is the multiplicative identity; for a square matrix, $AI = IA = A$.

Definition 1.6 *If A and X are $n \times n$ matrices and satisfy $AX = I$, then X is the inverse of A and is denoted by A^{-1} . If A^{-1} exists, then A is said to be nonsingular. Otherwise, A is singular. [24, p. 50]*

Definition 1.7 *A square matrix Q is said to be orthogonal if $Q^T Q = I$. [24, p. 69]*

With this definition and some algebra, it can be proved that the transpose of an orthogonal matrix is its own inverse. Since its transpose is well-defined, an orthogonal matrix is always nonsingular.

Some matrices have particular shapes:

Definition 1.8 *An $n \times n$ matrix A is upper triangular if $a_{ij} = 0$ for all $i > j$. (See Golub and Van Loan [24, Section 1.2.1].)*

Finally, adjacent rows or columns of a matrix can be conceptually combined together in groups:

Definition 1.9 *A stripe is a set of adjacent rows in a matrix. A colonnade is a set of adjacent columns. [46, p. 33]*

1.4.2 Matrix Multiplication

Matrix multiplication (Definition 1.4) is one of the most common operations on matrices, often used as a kernel operation for other algorithms. Multiplication can be generalized to blocks of the matrices. For the simplification of notation, let A , B , and C be $N \times N$ block matrices with $\ell \times \ell$ blocks. Then

$$C_{\alpha\beta} = \sum_{\gamma=1}^N A_{\alpha\gamma} B_{\gamma\beta} \quad (1.2)$$

for all $1 \leq \alpha \leq N$ and $1 \leq \beta \leq N$ [24, Section 1.3.5]. Instead of multiplying individual rows and columns, a block of C is computed by multiplying a stripe of A with a colonnade of B . The blocks of these matrices are square only because the notation is simpler; the matrices and their blocks could be rectangular.

This definition does not specify how $A_{\alpha\gamma} B_{\gamma\beta}$ should be multiplied. Equation 1.1 or Equation 1.2 is possible; the choice is the programmer's.

1.4.3 Solving a Linear System

A system of linear equations is commonly expressed in matrix form: given matrix A and vector b , one must solve $Ax = b$ for vector x . When A is a square, nonsingular matrix, the solution is unique and determinable.

The direct approach for solving $Ax = b$ is to factor A into other matrices that are

easily manipulated for finding x . For example, LU factorization factors A into two matrices L and U such that $A = LU$ where L is unit lower triangular and U is upper triangular [24, Section 3.2]. Solving $Ax = b$ for x then reduces to solving $LUx = b$. First, one solves $Ly = b$ for y by a process called forward substitution; then $Ux = y$ is solved for x by a process called backwards substitution [24, Section 3.1].

However, if A has more rows than columns, the system is said to be overdetermined; the problem is then framed in terms of finding an x such that the magnitude of the vector $Ax - b$ is minimized. This is the least squares problem [24, Section 5.3].

A common factorization for solving the least squares problem is QR factorization [24, Section 5.2]. QR factorization factors A into Q and R such that $A = QR$ where Q is orthogonal and R is upper triangular. The QR factorization can also be used to solve $Ax = b$ for x ; one computes $y = Q^T b$ (since Q is orthogonal) and then uses backwards substitution to solve $Rx = y$ for x .

Another use for the QR factorization is as a fundamental operation in the QR Algorithm [24, Chapter 7] to find the eigenvalues of a matrix.

1.4.4 Problems with Matrix Algorithms

Matrix algorithms must be written to handle potentially undefined operations such as division by zero and the square root of negative real numbers. An algorithm must either avoid these operations or generate an error when such an operation is attempted.

Another problem of matrix algorithms is computation error. Usually, finite-precision arithmetic is used to represent real numbers [24, Section 2.4.1]. Since that

representation is finite, the representation is generally inaccurate. Each operation (especially addition and subtraction) on these numbers can increase the error (see Golub and Van Loan [24, Section 2.4]), and matrix algorithms must be written so that errors are kept under control.

For example, some matrix factorizations (including LU factorization) require pivoting [24, Section 3.4] to keep errors in the factorization to a minimum. Without pivoting (or other special steps), these factorizations can yield grossly inaccurate results. Algorithms for QR factorization are much better behaved since the factoring is done with orthogonal matrices which do not yield large computational errors (see Section 5.4.3).

1.5 Traditional Matrix Storage

The main memory of a computer is commonly indexed as a one-dimensional array. Mapping a multi-dimensional structure like a matrix into main memory is a matter of determining a total, linear order for all of the elements.

Traditionally, matrices have been stored in main memory using row-major order or column-major order. For a matrix using row-major order (column-major order), every row (column) of the matrix is kept contiguous in the memory of the computer [29, p. 298, 299]. Figures 1.4 and 1.5 give the indices for these two orders for a 4×4 matrix.

Poor spacial locality results from, for example, traversing a row-major matrix by columns instead of rows [11, Section 2], yet column traversals are often necessary in iterative algorithms. (The same applies to row traversals of column-major matrices.)

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 1.4: Row-major indexing

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Figure 1.5: Column-major indexing

Better reuse of local memory (within a transfer block) is achieved by dealing with the matrix in terms of blocks as in Equation 1.2 for matrix multiplication. But since row- and column-major storage does not divide the matrix into contiguous blocks, a good algorithm must provide the blocking. One way to do this is by tiling the code [36, Section 20.4.3] (see Section 3.1).

1.6 The Quadtree Matrix

As suggested in Section 1.3.1, a divide-and-conquer data structure could be mapped into main memory using the divide-and-conquer paradigm. This section examines one such solution for a divide-and-conquer data structure for matrices.

1.6.1 Trees

Definition 1.10 *A tree is a finite set T of one or more nodes such that*

1. *there is one specially designated node called the root of the tree; and*

2. the remaining nodes (excluding the root) are partitioned into $m \geq 0$ disjoint sets T_1, T_2, \dots, T_m and each of these sets in turn is a tree. The trees T_1, T_2, \dots, T_m are called the subtrees of the root. [29, p. 308]

Definition 1.11 *The number of subtrees of a node is called the degree of that node. [29, p. 308]*

Definition 1.12 *A node of degree zero is called a terminal node, or sometimes a leaf. [29, p. 308]*

Definition 1.13 *The level of a node with respect to T is defined recursively: The level of $\text{root}(T)$ is zero, and the level of any other node is one higher than that node's level with respect to the subtree of $\text{root}(T)$ containing it. [29, p. 308]*

There are three types of trees of particular interest:

Definition 1.14 *A binary tree is a tree where the degree of every non-terminal node is exactly two. A quaternary tree, or quadtree, is a tree where the degree of every non-terminal node is exactly four. An octernary tree, or octtree, is a tree where the degree of every non-terminal node is exactly eight.*

Lewis and Denenberg [33, p. 101] define a useful term for binary trees that can be generalized for quadtrees and octtrees:

Definition 1.15 *All of the terminal nodes of a perfect tree are on the same level and all of the non-terminal nodes have maximum degree.*

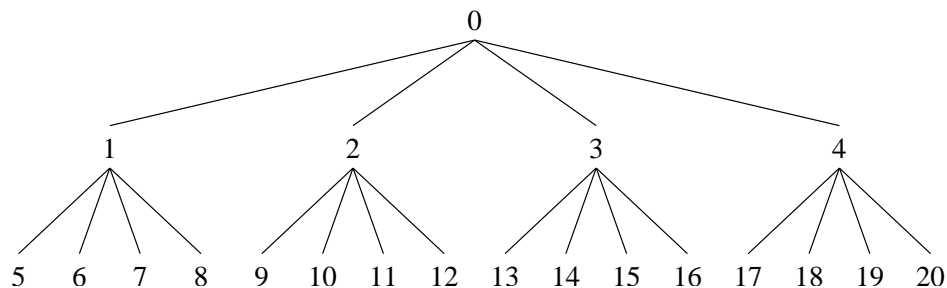


Figure 1.6: Level-order indexing of quaternary tree

1.6.2 Quadtree Indexings

A quadtree can be indexed in a variety of ways. One way to index a quadtree is to index the nodes of the tree, level by level:

Definition 1.16 *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 $4i + 4$.⁴ [47, p. 776]*

Figure 1.6 is the level-order indexing of a quadtree with three levels. The nodes of one level of the tree are indexed from left to right in increasing order. Level l is indexed before level $l + 1$.

Closely related to level-order indexing is Morton-order indexing.

Definition 1.17 *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 $4i + 3$. [47, p. 776]*

⁴Traditionally, the root of a binary tree has index 1 in level-order indexing [29, p. 401]; however, in trees with a higher degree (such as a quadtree), there are gaps in the indexing from one level to the next. A 0-indexed root works well for all trees.

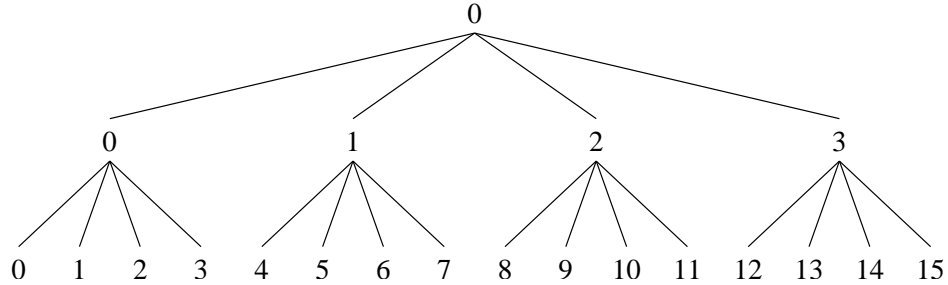


Figure 1.7: Morton-order indexing of quaternary tree

Figure 1.7 is the Morton-order indexing of a quadtrees with three levels. The main difference between these two indexings is that level order gives an index to every non-terminal node while Morton order indexes each level starting with index zero. The result is that the level-order indices on one level of the tree differ from the corresponding Morton-order indices by only a constant [47]—the number of nodes in the higher levels of the tree. Specifically, on level l , a Morton-order index and its corresponding level-order index differ by

$$\sum_{i=0}^{l-1} 4^i = \frac{4^l - 1}{3}. \quad (1.3)$$

This conversion makes these indexings easily interchangeable.

Perfect quadtrees make complete use of the indexing space of Morton-order and level-order indexing. Imperfect trees can still be indexed with level order and Morton order using the $4i+c$ computations, but the “missing” subtrees leave gaps in sequential indexing.

$$M_1 = \begin{bmatrix} 2.1 & 6.5 & 9.2 \\ 3.2 & 4.8 & 6.7 \\ 8.3 & 0.8 & 5.5 \end{bmatrix}$$

Figure 1.8: Matrix M_1

$$\text{pad}(M_1) = \begin{bmatrix} 2.1 & 6.5 & 9.2 & 0.0 \\ 3.2 & 4.8 & 6.7 & 0.0 \\ 8.3 & 0.8 & 5.5 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

Figure 1.9: Matrix M_1 padded

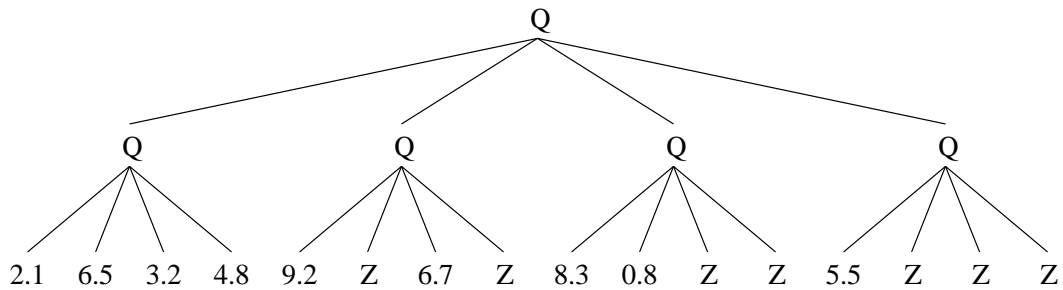


Figure 1.10: Quadtree matrix representation of $\text{pad}(M_1)$

1.6.3 Quadtree Matrix

Definition 1.18 *A quadtree 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. [46, p. 33]*

The quadrants of a quadtree matrix are indicated with the arrow operator \downarrow (e.g., $M_1 \downarrow \text{nw}$, $M_2 \downarrow \text{ne} \downarrow \text{nw}$); see Table 1.1.

The quadrant cleaving in the definition of the quadtree matrix suggests that the order of the quadtree matrix must be a power of two. The actual matrix may be padded with zeros to the south and east to bring its order up to a power of two. For example, the 3×3 matrix M_1 in Figure 1.8 is padded to a 4×4 matrix in Figure 1.9. The corresponding quadtree for M_1 is in Figure 1.10.

$$M_2 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 & 7 \end{bmatrix}$$

Figure 1.11: Matrix M_2

$$\text{pad}(M_2) = \left[\begin{array}{cccc|cccc} 1 & 2 & 3 & 4 & 5 & 0 & 0 & 0 \\ 6 & 7 & 8 & 9 & 1 & 0 & 0 & 0 \\ 2 & 3 & 4 & 5 & 6 & 0 & 0 & 0 \\ 7 & 8 & 9 & 1 & 2 & 0 & 0 & 0 \\ \hline 3 & 4 & 5 & 6 & 7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right]$$

Figure 1.12: Matrix M_2 padded

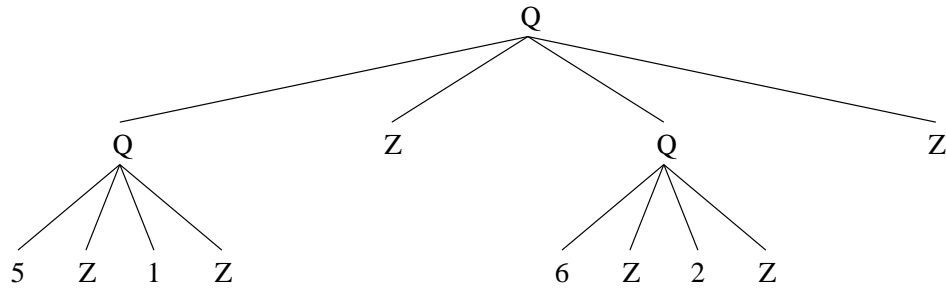


Figure 1.13: Quadtree matrix representation of $\text{pad}(M_2) \downarrow_{\text{ne}}$

While the padding could be nearly three-quarters of the padded matrix, the zero matrix and the tree structure of the quadtree matrix definition allows for a logarithmic savings of the space and time that is devoted to storing and processing the padding. For example, Figure 1.12 is the 8×8 padding of a 5×5 matrix M_2 given in Figure 1.11, and the northeast, southwest, and southeast quadrants are mostly padding. The potential savings is minimally demonstrated in the quadtree representation of the northeast quadrant of $\text{pad}(M_2)$, given in Figure 1.13.

In quadrants within a matrix (never at the top level), global padding may extend into its west colonnade or its north stripe:

Definition 1.19 *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.*

A perfect quadtree matrix would be stored in a perfect quadtree (see Definition 1.15), hence the name.

The padded matrices $\text{pad}(M_1)$ and $\text{pad}(M_2)$, Figures 1.9 and 1.12, respectively, have minority padding: padding only in the east colonnades and the south stripes of the top-level quadtree matrix matrix. The quadrant $\text{pad}(M_2) \downarrow \text{ne}$ has majority padding since the padding extends into the west colonnade of that quadrant. Similarly, $\text{pad}(M_2) \downarrow \text{sw}$ and $\text{pad}(M_2) \downarrow \text{se}$ have majority padding. $\text{pad}(M_2) \downarrow \text{nw}$ is a perfect quadtree matrix with no padding at all.

	Index Computation				
	Level	$4i + 1$	$4i + 2$	$4i + 3$	$4i + 4$
Morton		$4i + 0$	$4i + 1$	$4i + 2$	$4i + 3$
Z order		nw	ne	sw	se
W order		nw	sw	ne	se
U order		nw	sw	se	ne
C order		ne	nw	sw	se

Table 1.2: Correlation of quadrants to index computations

1.6.4 Quadtree Matrix Implementation

A quadtree matrix can be viewed as a quaternary tree: the general case (i.e., the sub-quadrants) are non-terminal nodes; scalars and zero matrices are terminal nodes.

A strict interpretation of the definition of a quadtree matrix (Definition 1.18) is often implemented as a linked data structure with the zero matrix represented as a special terminal node, but as noted in Section 1.3.2, linked data structures suffer from data-locality deficiencies.

For better data locality, the nodes of a quadtree matrix can be put into an array (or arrays) of contiguous memory using an indexing for the quaternary tree. There are twenty-four (i.e., $4! = 24$) different ways to correlate quadrant names (i.e., northwest, northeast, southwest, and southeast) and index computations (e.g., $4i+0$, $4i+1$, etc.). Table 1.2 lists four different ways to correlate quadrants and index computations. Often the application determines the best correlation. Z order is favored in the graphics community because pictures tend to be wide (more columns than rows). Then, if there were less than half as many rows as columns, it would be unnecessary to allocate space for the southern rows. Matrices often have more rows than columns,

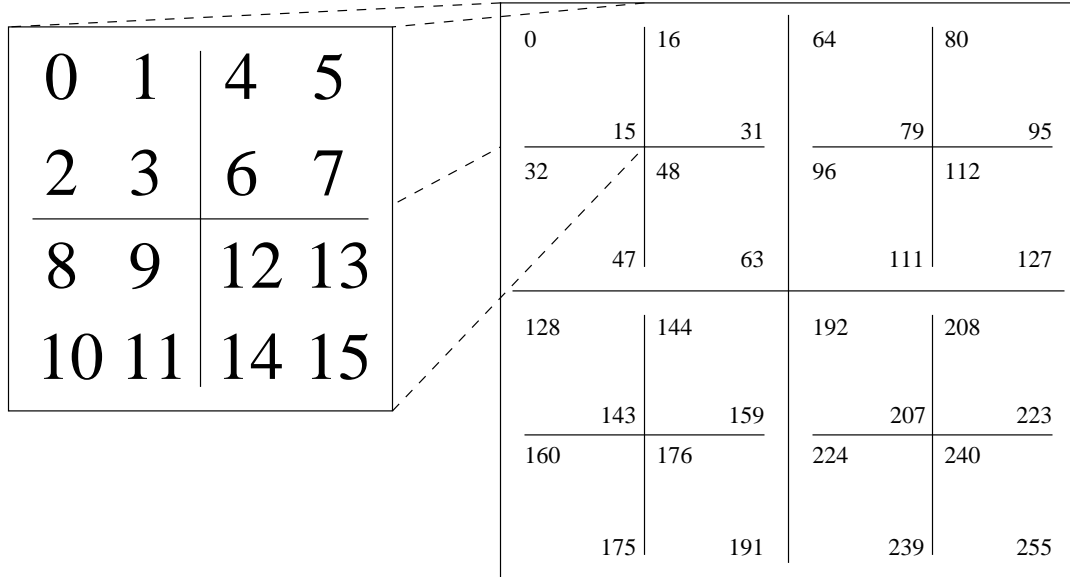


Figure 1.14: Morton ordering of a 16×16 matrix with Z order

so \mathcal{H} order is useful for matrices. Either order is equally useful for square matrices.

Figure 1.14 is the Morton order indexing of a 16×16 matrix using the Z order.

With matrices padded to the south and east, it is best to index the southeast quadrants last. The padding will always be greatest in this quadrant, and the southeast most element of the unpadded matrix determines the amount of space that must actually be allocated since the padding that follows it is not necessary for the indexing to work correctly.

For example, consider the padded matrix $\text{pad}(M_2)$ of Figure 1.12. A perfect 8×8 matrix would require memory for 64 scalars; but with Z order, $\text{pad}(M_2)$ only needs memory for $48 + 1 = 49$ scalars since the Morton-order index of the southeast most element is 48. Only zeros are found at Morton indices 49 through 63. Unlike the

padding in the northeast and southwest quadrants of $\text{pad}(M_2)$, this padding in the southeast need not be allocated in the computer’s memory. In general, the padding of the padded matrix might be asymptotically three-quarters of the matrix; but since the majority of the southeast quadrant would not be allocated, the padding will be only two-thirds, not three-quarters, of the total memory allocated. Avoiding the rest of the padding can be handled by decorating the matrices; as long as this memory is avoided, it only consumes address space, not valuable memory space.

1.6.5 Decorating Quadtree Matrices

An initial concern with this sequential representation is the gaps left by the indexing of imperfect trees (as noted at the end of Section 1.6.2). The gaps in the address space caused by padding blocks makes the corresponding memory space unnecessary except as internal padding. If an algorithm avoids these gaps, they only waste address space and cheap, slow memory. These blocks will never be called into expensive, high-level cache.

To avoid this padding, quadtree matrix algorithms must recognize zero matrices high in the quadtree. Decorations can be used to represent zero matrices:

Definition 1.20 *An annotation describing the contents of a quadrant of a quadtree matrix is called a decoration. (See Wise [46, pp. 66, 67].)*

For this work, there are four basic decorations: **zero**, **identity**, **dense**, and **unknown**.

- The **unknown** decoration (denoted as ‘Q’ in figures) indicates that the quadrant

has a mixture of dense, zero, and identity matrices, so the testing of decorations must continue as the sub-quadrants are traversed.

- The **zero** decoration ('Z') indicates that the quadrant is homogeneously zero. The algebra of the zero matrix can be used to control algorithms without descending into the quadrants.
- The **identity** decoration ('I') indicates that the quadrant is an identity matrix. Similar to the **zero** decoration, the algebra of the identity matrix can be used to control algorithms without descending into the quadrant.
- The **dense** decoration ('D') says that the quadrant is completely dense. There are not enough zero or identity blocks to test for them. All testing of decorations can be suspended for this quadrant.

1.6.6 Storage Arrays

Since the decorations and elements of the matrix are different data types, two arrays are allocated:

- One array stores the decorations. The decorations are found in the non-terminal nodes of the quadtree, so the decoration array uses level-order indexing to store all of these nodes compactly. Given just the decorations in Section 1.6.5, two bits are sufficient to represent the decorations; eight bits were used since one byte is the smallest unit of memory easily accessed.
- One array stores the matrix elements. All of these elements are found at the leaf level of the quadtree, and so Morton-order indexing is used to store just

Q	D	Q	Q	Q
0	1	2	3	4

Figure 1.15: Level-order storage of $\text{pad}(M_1)$ decoration (cf. Figure 1.9)

2.1	6.5	3.2	4.8	9.2	0.0	6.7	0.0	8.3	0.8	0.0	0.0	5.5
0	1	2	3	4	5	6	7	8	9	10	11	12

Figure 1.16: Morton-order storage of $\text{pad}(M_1)$ scalars

these leaf elements. Each of these nodes requires eight bytes for double precision floating-point numbers.

The decoration matrix takes up comparatively little space. The element array at eight bytes for each element takes up twenty-four times as much memory as the decoration array at eight bits per decoration.

Level-order indices are needed to access the decoration array; Morton-order indices are needed to access the element array. Rather than traversing the nonterminal nodes with level-order indices and repeatedly converting them to Morton-order indices to access the element array, the pointer for the element array is offset using Equation 1.3. This pointer works as if the element array were allocated as level-order; but, in fact, the indices for nonterminal nodes should not be and never are used on the element array. This offset computation is done only once at the beginning of an algorithm.

Figures 1.15 and 1.17 depict the storage of the decoration trees for matrices $\text{pad}(M_1)$ and $\text{pad}(M_2) \downarrow \text{ne}$ from Figures 1.9 and 1.12, respectively. Both of these arrays use level-order indexing (in Z order) for the arrays; for $\text{pad}(M_2) \downarrow \text{ne}$, only the

...	Q	...	Q	Z	Q	Z	...
...	2	...	9	10	11	12	...

Figure 1.17: Level-order storage of $\text{pad}(M_2 \downarrow \mathbf{ne})$ decoration (cf. Figure 1.12)

...	5	0	1	0	⊥	⊥	⊥	⊥	6	0	2	0	⊥	⊥	⊥	⊥	...
...	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	...

Figure 1.18: Morton-order storage of $\text{pad}(M_2 \downarrow \mathbf{ne})$ scalars

relevant portions of the array are given for that quadrant. Similarly, Figures 1.16 and 1.18 depict the storage of the scalars of the same matrices. These arrays are indexed using Morton-order indices (also in Z order), starting the array at index 0 for the northwest most elements. There is no need to represent the zero scalars in the scalar array because of the **zero** decoration, so that memory is not changed or even accessed.

1.6.7 A Divide-and-Conquer Data Structure

It is important to note how Morton-order and level-order indexing maps quadrants into an array: quadrants are in contiguous memory. This mapping into contiguous memory is done for *every* quadrant at *every* level of the quadtree. Furthermore, *every* level of the memory hierarchy on *every* machine will have some level of the quadtree matrix that fits into a transfer block of that level without using any special knowledge of the sizes of the transfer blocks.

Thus, in a practical sense, *a transfer block is a quadrant.*

Consequently, whatever is good for quadrants is good for transfer blocks. The particulars of the transfer blocks are irrelevant. As a programmer programs for quadrants, the programmer programs for all memory hierarchies.

1.7 Row Major Versus Morton Order

A high-performance algorithm should work for every level of the memory hierarchy. Row-major solutions⁵ for handling the memory hierarchy, like tiling (see Sections 1.5 and 3.1), must be applied to the code for every level of the hierarchy. Tile the code once for primary cache and then for secondary cache and once again for virtual memory. When a new level is added, the code must be tiled again for the new level. When a level of the existing hierarchy is reconfigured (e.g., by adding more RAM), the tiling must change for the reconfigured level. When the code is ported to a new machine, the hierarchy is different, and so the tiling must be redone. All of these tiling changes mean changes to the code which necessitates recompiling. Furthermore, for each of these changes, the programmer or optimizing compiler must know the particulars about the memory hierarchy.

Solutions for quadtree matrices stored in Morton order⁶ are simpler. A quadrant-friendly algorithm is written *once*. Since the quadrants are transfer blocks, a quadrant-friendly algorithm is implicitly friendly to transfer blocks. A recursive algorithm using quadrants at all levels of the quadtree will fit them—at some level—into the transfer blocks at each level of the memory hierarchy. When a level of the memory hierarchy is added or reconfigured, the mapping between quadrants and transfer blocks takes

⁵Row major is equivalent to column major for these comments.

⁶Similarly, Morton order and level order are equivalent for these comments.

care of the added or reconfigured level. When the code is ported to a new machine, the mapping between quadrants and transfer blocks handles the new hierarchy. The code is recompiled only for new processors, not for new memory hierarchies. Furthermore, since the mapping does not require any parameters, the programmer and compiler can be oblivious to the particulars of the memory hierarchy (i.e., being “cache-oblivious” [22]).

1.8 Related Research

In his dissertation, Beckman [6] presents a (nearly pure) functional implementation of LU factorization for sparse matrices using a linked quadtree. He describes a distributed, shared-heap storage manager and a run-time system to execute LU factorization on sparse matrices in parallel on a shared-memory machine. The storage manager and run-time system are quite successful, but the performance of LU factorization is disappointing. Performance seems to be affected adversely by the poor compilation of recursion, the low payoff (i.e., one or two floating point operations) at each base case, and locality problems using a linked structure.

Wise [46] describes a more abstract and elaborate presentation of LU factorization. He concentrates on exact arithmetic matrix inversion (for use with discrete objects like integers and polynomials) and stresses the basic algebra of the algorithm that lends itself to a functional formulation. Wise uses decorations for directing the selection of a pivot matrix.

G. M. Morton introduced his indexing originally for geodetic databases [35]. The indexing is old and has been rediscovered several times. Samet [39] and others in the

graphics community use Morton-order indexing for storing and processing images.

Strassen [43] and Spieß [41] use a block decomposition of matrices to reduce the overall number of scalar multiplications for matrix multiplication. While reducing the flop count from $O(n^3)$ to $O(n^{\lg 7})$, their algorithms requires extra space.

Chatterjee et al. explore some of the programming and compiler optimizations that can be done to make matrix multiplication run faster when using Morton order to store the scalar matrix [12, 11]. They chose a hybrid representation, using Morton order on the upper-level blocks and column-major for the base-case blocks. This allowed them to use BLAS routines (see below) for the base case. They found the hybrid awkward: either algorithms had to be rewritten for it or the data must be converted to and from other representations at a prohibitive cost.

Gustavson considers recursion and blocking for dense linear algebra [25]; Elmroth and Gustavson consider QR factorization in particular [18]. However, their recursive algorithm for QR factorization is in terms of columns (a single column is the base case for their recursion).

Golub, Plemmons, and Sameh [23] use a blocked algorithm for QR factorization in the context of the least-squares problem. Their matrices are sparse and patterned (block angular), obtained from geodetic computations.

Frigo, Leiserson, Prokop, and Ramachandran [22] study what they call *cache-oblivious* algorithms. Cache-oblivious algorithms are told *nothing* about the memory parameters of the machine that they run on, yet they still perform well with respect to the memory hierarchy. Their solution does not block the data representation, but it uses recursion on the existing representations (row- or column-major in the case of

matrices). By their definition, the algorithms developed here are cache-oblivious.

The BLAS (Basic Linear Algebra Subprograms) library [32, 17, 16] was written to implement basic matrix and vector operations (like matrix multiplication) as efficiently as possible for column- and row-major storage in both uniprocessor and parallel versions. The routines in this library are often tuned by manufacturers for their particular machines for even better performance.

LAPACK [2] is a high-performance library that implements high-level matrix algorithms like matrix factorizations. Many LAPACK algorithms use BLAS routines; other algorithms are written from scratch. The general algorithms are tuned for column- and row-major storage.

More recent work has been done on writing compilers that tune algorithms for traditional storage automatically. Both PHiPAC [7] and ATLAS [45] gather information (explicitly or through experimentation) to generate efficient code tuned for a specific machine. However, ATLAS only blocks for two levels of cache [45, p. 11]⁷.

1.9 New Contributions

The duals for multiplication (see Section 3.2.1) for good cache behavior were first presented in Frens and Wise [20]. This dissertation is the first to analytically prove its good memory behavior (in Section 3.2.2).

Quadtree decorations themselves are not new. This work develops the use of decorations for matrix multiplication and QR factorization. In particular, the first

⁷More recent reports about ATLAS available on the Web indicate that this has not changed since 1998 when this paper was written.

and last functions for matrix multiplication (see Sections 3.4.4 and 3.4.5) are new.

While Elmroth and Gustavson [18] present a similar QR factorization, their basic algorithmic unit (i.e., their base case) is still a column, not a block. The algorithms presented in Chapter 5 work on the quadrants of the matrix; this leads to a similar, but new and distinctive algorithm.

The parallel dispatch for parallel matrix multiplication was first presented in Frens and Wise [20]. The parallel dispatch for QR factorization is new.

1.10 Road Map

The next chapter lays out the playing field for this work. This includes discussions about this work as a proof-of-concept and the high-performance environment used to run these experiments including a description of the machines used to perform tests and a discussion of the compiler and timing issues involved with those tests.

Chapters 3 and 4 cover matrix multiplication. Chapter 3 covers the uniprocessor quadtree algorithm, including a proof of memory efficiency; Chapter 4 discusses a parallel implementation of this algorithm. The performances of both the uniprocessor and parallel algorithms are compared to the performances of comparable BLAS code.

Chapters 5 and 6 are organized similarly for QR factorization. Chapter 5 explores a quadtree matrix algorithm for QR factorization; the following chapter presents a parallel algorithm. The uniprocessor and parallel performances of the quadtree algorithms are compared to the performances of comparable LAPACK programs.

The final chapter offers conclusions and points to future work.

2

High-Performance Environment

2.1 Proof of Concept

Any technology goes through several stages before it gains wide acceptance. Using functional programming and divide-and-conquer to develop high-performance algorithms is in an early stage. The purpose of this work is to give a proof-of-concept for the thesis that functional programming is an effective tool for solving high-performance problems. This proof-of-concept consists of two matrix problems: matrix multiplication and QR factorization. Matrix multiplication is simple and fundamental; QR factorization is more complex and is an important algorithm. Both classic algorithms for these problems have favorable error analyses [26, Sections 1.14.2, 3.5, and 18.5] so the quadtree algorithms were developed without consideration for error handling.

This work establishes an intermediate stage for a compiler. Ideally, a programmer would not write this code directly but instead would write pure functional code that

would be translated by a compiler with a knowledge of optimizing memory management. This work does not deal with such a translation, but instead concerns itself with what the target of such a translation should be.

2.2 Programming Liberties

Several liberties are taken with functional programming in order to compare performance with established high-performance computing results. Most functional programming languages depend on linked memory and a run-time garbage collector; this often hurts the performance of functional programs due to the time for collection and because linked structures have poor locality in general. If side-effects are permitted (strictly forbidden in pure functional programming), then the need for a garbage collector can be reduced, perhaps eliminated. The programs of this work manage memory on their own, allocating blocks of memory for the matrices and side-effecting the data in place. All matrices are uniquely referenced, so the side-effects are always safe. No new memory is allocated after the initial allocations, so there is no need for a garbage collector.

The algorithms were written in C to take advantage of the manufactures' optimizing compilers for their architectures. Supporting and driver code was written in C++.

The algorithms were also simplified by assuming that all matrices are square. In particular, this simplified the parallel dispatch considerations in Chapters 4 and 6. Rectangular matrices could be handled by extending the ideas presented in those chapters.

System Property	Value
Number of processors	10
Type of processors	R8000 (MIPS)
Clock speed	75 megahertz
Virtual memory	2 gigabytes
RAM	2 gigabytes shared by all processors
Secondary cache	4 megabytes per processor
Instruction cache	16 kilobytes per processor
Data cache	16 kilobytes per processor
Maximum mflop/s	300

The floating point unit of an R8000 is not connected to the primary cache and does not have one of its own; it is connected to the secondary cache [40, Section 2.2].

Table 2.1: System parameters of Power Challenge

2.3 Machines

Four machines were used to run timing experiments: an SGI Power Challenge (Table 2.1), an SGI Octane (Table 2.2), a Sun Enterprise 450 Model 4400 (Table 2.3), and a Sun Ultra 5/10 (Table 2.4).

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 were minimal.

2.4 Compiler Issues

2.4.1 Compilers and Libraries

The MIPSpro compiler was used on the SGI machines. The manufacturer provided their own BLAS libraries; LAPACK was compiled locally. The Sun Workshop

System Property	Value
Number of processors	1
Type of processor	R10000 (MIPS)
Clock speed	195 megahertz
Virtual memory	39.1 gigabytes
RAM	128 megabytes
Secondary cache	1 megabyte
Instruction cache	32 kilobytes
Data cache	32 kilobytes
Maximum mflop/s	390

Table 2.2: System parameters of Octane

System Property	Value
Number of processors	4
Type of processors	UltraSparc II
Clock speed	400 megahertz
Virtual memory	5.4 gigabytes, shared
RAM	2 gigabytes, shared
Secondary cache	4 megabytes per processor
Instruction cache	16 kilobytes per processor
Data cache	16 kilobytes per processor
Maximum mflop/s	400

Table 2.3: System parameters of Enterprise 450

System Property	Value
Number of processors	1
Type of processor	UltraSparc Ili
Clock speed	440 megahertz
Virtual memory	1.1 gigabytes
RAM	256 megabytes
Secondary cache	4 megabytes
Instruction cache	16 kilobytes
Data cache	16 kilobytes
Maximum mflop/s	440

Table 2.4: System parameters of Ultra 5/10

Compiler 5.0 was used with the Sun Performance Library 2.0, which supplies both the BLAS and LAPACK libraries, on the Sun machines.

2.4.2 Stride

The BLAS and LAPACK functions were tested with column major matrices. To avoid most problems with stride (see Golub and Van Loan [24, Section 1.4.4]), every array was allocated with an odd stride, even if the order of the matrix was even.

2.4.3 Recursive Code

Since most optimizing compilers for C do not optimize recursion well, some optimizations for the quadtree algorithms were done by hand. Any of these hand-optimizations could be done by a recursion-aware optimizing compiler.

One hand-optimization was to store array pointers in global variables since they change infrequently (and only in the QR factorization). This greatly reduced the

overhead incurred with the recursion.

Another hand-optimization is the unfolding of recursive code. Loops are routinely unrolled by optimizing compilers to take advantage of superscalar architectures [36, Section 17.4.5]; analogously, recursive functions should be unfolded [10]. Each unfolding also eliminates one level of function calls, avoiding all of the function call overhead. Since optimizing compilers for C do not unfold recursive code automatically, the unfolding was done by hand. (See Section 3.4.2.)

2.4.4 Compiler Flags

All programs were compiled with the highest optimization turned on. See Tables 2.5 and 2.6 for the specific optimizations. No effort was taken to eliminate redundant flags (e.g., on the SGIs `-64` implied `-mips4`, but both flags were used anyway).

2.4.5 Shared Memory and Parallel Dispatch

Shared memory for the matrices was allocated using the function `mmap()`. Parallel dispatch was done with the function `fork()`. This allowed the matrix arrays to be shared while each forked process had its own runtime stack and global variables.

Optimization flag	Effect
-Ofast=ip21	used on Power Challenge, maximizes performance for the ip21 architecture
-Ofast=ip30	used on Octane, maximizes performance for the ip30 architecture
-64	generates 64-bit objects
-mips4	generates code using the full MIPS IV instruction set
-r8000	used on Power Challenge, specifies the processor for scheduling code
-r10000	used on Octane, specifies the processor for scheduling code
-SWP:=ON	turns on software pipelining
-OPT:alias=RESTRICT	specifies that distinct pointers are assumed to point to distinct, non-overlapping objects
-IPA	turns on the inter-procedural analyzer

Table 2.5: Compiler optimizations for SGIs

Optimization flag	Effect
-fast	maximizes optimizations
-xarch=v8plus	maximizes performance for the architecture

Table 2.6: Compiler optimizations for Suns

2.5 Timing Issues

2.5.1 Measuring Performance

The most important operation in high-performance algorithms are those involving floating point numbers:

Definition 2.1 *A flop is a floating point operation.* [24, Section 1.2.4]

Each multiplication and each addition counts as one flop. Performance is measured in terms of the number of flops done per second. Often, this number is measured in the millions, so the speed of the algorithm is measured in megaflops per second, abbreviated *mflop/s*.

The performance of parallel algorithms are measured in terms of speed-up:

Definition 2.2 *A parallel algorithm for a particular problem achieves speed-up S if*

$$S = T_{seq}/T_{par}$$

where T_{par} is the time required for execution of the parallel program and T_{seq} is the time required by one processor when the best uniprocessor procedure is used. [24, Section 6.1.6]

Suppose p is the number of processors used for a parallel algorithm. Ideally, the parallel algorithm should give a speed-up equal to the number of processors, $S = p$; while this is not always possible, parallel algorithms are evaluated in terms of how close its speed-up is to p . Occasionally parallel algorithms will demonstrate super-linear speed-up where $S > p$ because of increased caching on the multiple processors.

2.5.2 Machine Solutions

The algorithms were timed using `getrusage()` on the Sun machines and a high-resolution timer available through the function `syssgi()` on the SGI machines. These functions measure the *processor time* consumed, not the wall-clock time. Only the algorithms themselves were measured; initialization or clean up code was *not* timed using these timers. These processor timers were used for all uniprocessor results on all machines for all algorithms.

These processor timers were also used to compute parallel speed-up. The drivers were written to measure the time spent by the original process (assumed to have as much or more work than the other parallel processes). This technique was used on the Power Challenge for all tests and for the quadtree matrix algorithms on the Enterprise 450.

The BLAS and LAPACK routines on the Enterprise 450 use threads to implement parallelism. The processor timers would measure the total time spent by *all* threads, making the results useless for speed-up calculations. Instead, a wall-clock timer `gethrtime()` was used to measure the wall-clock time. These wall-clock times were used *only* to calculate the speed-ups on the Enterprise 450, and so those results are a bit sporadic. Uniprocessor results on this machine were still obtained using a processor timer.

3

Matrix Multiplication

Matrix multiplication is a kernel operation in most matrix algorithms. It also teaches much about writing and implementing algorithms for quadtree matrices.

As noted in the previous chapter, all matrices are now assumed to be square.

3.1 Row-Major Multiplication

Matrix multiplication (Equation 1.1) can be written as three nested loops as in Figure 3.1. In this algorithm, matrix A is traversed by rows and B is traversed by columns. When the matrices are stored in row-major order and n is sufficiently large,

```
nestedLoops (int n, RMatrix c, RMatrix a, RMatrix b) {
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];
}
```

Figure 3.1: Nested loops for inner-product multiplication

```

blockedNestedLoops (int n, int N,
                    RMMatrix c, RMMatrix a, RMMatrix b) {
    int ell = n / N;
    for (int I = 0; I < N; I++)
        for (int J = 0; J < N; J++)
            for (int K = 0; K < N; K++)
                for (int ii = I*ell; ii < I*(ell+1); ii++)
                    for (int jj = J*ell; jj < J*(ell+1); jj++)
                        for (int kk = K*ell; kk < K*(ell+1); kk++)
                            c[ii][jj] += a[ii][kk] * b[kk][jj];
}

```

Figure 3.2: Blocked nested loops for multiplication

the column traversal of B in the inner loop will likely generate a costly memory miss on each iteration of the innermost loop.

This problem is not peculiar to row-major storage. If the matrices were stored in column-major order, the same algorithm would generate analogous memory misses on A . The problem is also not peculiar to this ordering of the loops. The nested loops can be ordered in six different ways [24, Section 1.1.11], and each ordering has analogous memory access problems on A , B , or C .

This problem is mollified by traversing the matrices by blocks. The loops in Figure 3.1 are modified through a process known as tiling [36, Section 20.4.3]. Equation 1.2 gives the basic idea: each matrix is divided into $N \times N$ blocks to yield an $\ell \times \ell$ block matrix where $\ell = n/N$ [24, Section 1.4.7]. The resulting algorithm with one level of tiling is in Figure 3.2.

The size of a matrix block (i.e., ℓ) is determined by the size of a transfer block. Since the transfer blocks of each level of the memory hierarchy have different sizes, one level of tiling (i.e., one ℓ) works for only one level of the memory hierarchy. So

tiling is repeated for each level of the hierarchy.

Often an optimizing compiler will do the tiling for the programmer, incorporating knowledge of the size of a transfer block of the target machine to pick an appropriate ℓ for each level of the memory hierarchy [7, 45].

3.2 Quadtree Matrices

A quadtree matrix stored in a serial array using Morton-order indexing is blocked in its storage at each level of the quadtree matrix. By programming for the quadrants of a quadtree matrix, a programmer deals with transfer blocks on all levels of the memory hierarchy.

3.2.1 Two Versions of Matrix Multiplication

Matrix multiplication for a quadtree matrix is given in Figure 3.3 [20]; it is equivalent to Equation 1.2 with $N = 2$. Data locality is achieved by reusing quadrants in one step to the next: $C \downarrow ne$ is reused on the second step of the function, $B \downarrow se$ on the third, $C \downarrow se$ on the fourth, and so on. Since C is written as well as read, it may generate more costly memory traffic (depending on the write-back properties of memory); so every effort is made to reuse quadrants of C . This is also beneficial when planning the parallel dispatch (see the next chapter).

However, the algorithm in Figure 3.3 does *not* actually reuse memory blocks. While it appears that a quadrant of one matrix is reused between any two consecutive statements, the sharing does not continue through the sub-quadrants. For example,

```

quadtreeMultiply (QMatrix C, QMatrix A, QMatrix B) {
  if (scalar (C))
    C += A * B
  else {
    quadtreeMultiply (ne(C), nw(A), ne(B));
    quadtreeMultiply (ne(C), ne(A), se(B));
    quadtreeMultiply (se(C), se(A), se(B));
    quadtreeMultiply (se(C), sw(A), ne(B));
    quadtreeMultiply (sw(C), sw(A), nw(B));
    quadtreeMultiply (sw(C), se(A), sw(B));
    quadtreeMultiply (nw(C), ne(A), sw(B));
    quadtreeMultiply (nw(C), nw(A), nw(B));
  }
}

```

Figure 3.3: Cache-poor quadtree matrix multiplication

it appears that $C \downarrow ne$ is shared from the first to second statements; however, the first recursive call accesses $C \downarrow ne \downarrow nw$ last while the second recursive call accesses $C \downarrow ne \downarrow ne$ first—there is no real sharing on the next level down. When the quadrants of $C \downarrow ne$ are sufficiently large, costly memory misses are triggered when the programmer’s intent was to share quadrants and avoid the memory misses.

This problem is solved by writing dual algorithms that complement one another with respect to their access patterns. These algorithms are presented in Figure 3.4 [20]. The two algorithms have been named *up* and *down* to suggest their dual nature and to avoid the overloading of terms.¹

These two functions working together produce good memory behavior. They have been crafted as complements to each other. Consider again the first two statements

¹The names have as much significance as the up and down spin of electrons—none.

```

dnMult (QTMatrix C,
        QTMatrix A, QTMatrix B) {
  if (scalar (C))
    C += A * B;
  else {
    dnMult (ne(C), nw(A), ne(B));
    upMult (ne(C), ne(A), se(B));
    dnMult (se(C), se(A), se(B));
    upMult (se(C), sw(A), ne(B));
    upMult (sw(C), sw(A), nw(B));
    dnMult (sw(C), se(A), sw(B));
    upMult (nw(C), ne(A), sw(B));
    dnMult (nw(C), nw(A), nw(B));
  }
}

upMult (QTMatrix C,
        QTMatrix A, QTMatrix B) {
  if (scalar (C))
    C += A * B;
  else {
    upMult (nw(C), nw(A), nw(B));
    dnMult (nw(C), ne(A), sw(B));
    upMult (sw(C), se(A), sw(B));
    dnMult (sw(C), sw(A), nw(B));
    dnMult (se(C), sw(A), ne(B));
    upMult (se(C), se(A), se(B));
    dnMult (ne(C), ne(A), se(B));
    upMult (ne(C), nw(A), ne(B));
  }
}

```

Figure 3.4: Cache-friendly quadtree multiplication

of `dnMult()`:

```

dnMult (ne(C), nw(A), ne(B)); // #1
upMult (ne(C), ne(A), se(B)); // #2

```

Statement #1, through the recursive call to `dnMult()`, will access $C \downarrow ne \downarrow nw$ *last*. Statement #2, through the call to `upMult()`, accesses this *same* quadrant *first*. Similar observations can be made for any two consecutive statements in both `dnMult()` and `upMult()`. The sharing is not limited to these two levels; it extends all the way down through the quadrants of the matrices, and consequently, all throughout the memory hierarchy. This is proved in the next section.

3.2.2 Analytical Proof of Good Memory Behavior

The base cases of `upMult()` and `dnMult()` vacuously reuse memory since there is no memory to share. So only the recursive cases must be examined to prove their

good memory behavior. First, these lemmas describe how the functions treat the quadrants of the three matrices:

Lemma 3.1 *Suppose that A , B , and C are an $2^p \times 2^p$ matrices where $p \geq 1$. Then the following claims are true:*

1. *The function $\text{upMult}()$ accesses the northwest-most quadrants of C , the northwest-most quadrants of A , and the northwest-most quadrants of B first.*
2. *The function $\text{dnMult}()$ accesses the northeast-most quadrants of C , the northwest-most quadrants of A , and the northeast-most quadrants of B first.*
3. *The function $\text{upMult}()$ accesses the northeast-most quadrants of C , the northwest-most quadrants of A , and the northeast-most quadrants of B last.*
4. *The function $\text{dnMult}()$ accesses the northwest-most quadrants of C , the northwest-most quadrants of A , and the northwest-most quadrants of B last.*

Proof by induction on the depth of the recursion (which is logarithmic of the order of the matrix).

Base case. The base case of this induction is a 2×2 matrix ($p = 1$). Consider Claim #1. The first recursive call in $\text{upMult}()$ is $C_{\downarrow\text{nw}+} = A_{\downarrow\text{nw}} \cdot B_{\downarrow\text{nw}}$. Since each of these quadrants are scalars, they are the northwest-most quadrants for all three matrices.

The other base-case claims are proved similarly.

Induction case. For the induction case $p > 1$, the induction hypothesis assumes that the lemma is true for quadrants of size $2^{p-1} \times 2^{p-1}$. Then for Claim #1,

it is sufficient to observe that `upMult()` accesses the northwest quadrants of C , A , and B first. Since these accesses are done recursively with `upMult()` itself and the recursive quadrants have size $2^{p-1} \times 2^{p-1}$ satisfying the conditions of the induction hypothesis, all of the northwest-most quadrants of C , A , and B are accessed first.

The other induction-case claims are proved similarly. □

The step from one recursive call to the next in the recursive calls of the algorithms of Figure 3.4 will simply be called a *transition*.

Theorem 3.1 *A quadrant at every level of one of the three quadtree matrices is shared in all of fourteen transitions of both `upMult()` and `dnMult()`.*

Proof Consider the first transition of `upMult()`. $C \downarrow \text{nw}$ is shared. Claim #3 of Lemma 3.1 says that the northeast-most quadrants of $C \downarrow \text{nw}$ will be the last accessed. Claim #2 of the same lemma says that these same quadrants will be the first accessed by `dnMult()`. Thus all of these northeast-most quadrants of $C \downarrow \text{nw}$ are shared on the first transition.

The other transitions of `upMult()` and all of the transitions of `dnMult()` are proved similarly. □

The theorem leads to this important corollary:

Corollary 3.1 *A transfer block at any level of the memory hierarchy is shared in all of the fourteen transitions of `upMult()` and `dnMult()`.*

Proof As established in Section 1.7, transfer blocks are the same as quadrants. Substitute “transfer block” for “quadrant” and “memory hierarchy” for “quadtrees matrix” in Theorem 3.1. □

This solution differs from tiling in several very significant ways. These dual functions are written once while tiling must be done for each level of the memory hierarchy. The dual quadtree-matrix algorithms work for *all* levels of the hierarchy regardless of size and regardless of any changes made to the levels of the hierarchy without no special knowledge about the memory hierarchy. This means that the code does not have to be rewritten or even recompiled when the hierarchy changes.

3.3 Successful Matrix Multiplication

Matrix multiplication is a well behaved operation with little to cause it to fail. There are no undefined operations in matrix multiplication, so the only problem is the error analysis of matrix multiplication.

Higham notes that the versions of matrix multiplication that traverse the matrices by rows and columns all have the same error analysis, an error proportional to the product of the norms of A and B [26, Section 3.5]. The issues involved in this analysis are no worse for tiled versions of these algorithms, for `dgemm()` of BLAS (see Section 3.4.1), and for the quadtree matrix algorithm.

3.4 Compiler and Implementation Issues

See also the compiler issues discussed in Section 2.4.

3.4.1 BLAS

BLAS (Section 1.8) is a library of basic matrix (and vector) algorithms. The BLAS function for multiplying general matrices of double precision floating point numbers is called `dgemm()`. The performance of this function is compared to the performance of the quadtree matrix multiplication in the following section. The hand-tuned BLAS from the manufacturers were used on all machines.

3.4.2 Unfolding the Base Case

For recursive code like matrix multiplication in Figure 3.4, each unfolding (see Section 2.4.3) is exponential in the number of recursive calls. One unfolding [10] of one multiplication routine leads to a base case of $8^1 = 8$ multiplications; another unfolding yields $8^2 = 64$ multiplications; yet another yields $8^3 = 512$ multiplications. The number of multiplication statements can be reduced by rolling the code into a loop. Figures 3.5 and 3.6 demonstrates a rolling of an unfolded 2×2 base case. (This is for illustration only; a 2×2 base case is too small for rolling to be effective in practice.)

Unfolding is useful not only for superscalar architectures (Section 2.4.3), but it also eliminates many function calls and the overhead associated with them. This is apparent in matrix multiplication. If multiplication is viewed as an octtree (each

```

// iC, iA, & iB contain Morton index of northwest
// corner of current block for respective matrices
C[iC+0] += A[iA+0] * B[iB+0] + A[iA+1] * B[iB+2];
C[iC+1] += A[iA+0] * B[iB+1] + A[iA+1] * B[iB+3];
C[iC+2] += A[iA+2] * B[iB+0] + A[iA+3] * B[iB+2];
C[iC+3] += A[iA+2] * B[iB+1] + A[iA+3] * B[iB+3];

```

Figure 3.5: Original unfolded 2×2 base case

```

// iC, iA, & iB contain Morton index of northwest
// corner of current block for respective matrices
for (row = 0; row < 2; row++) {
    C[iC+0] += A[iA+0] * B[iB+0] + A[iA+1] * B[iB+2];
    C[iC+1] += A[iA+0] * B[iB+1] + A[iA+1] * B[iB+3];
    switch (row) {
        case 0: iC += 2; iA += 2; break;
        case 1: iC -= 2; iA -= 2; break;
    }
}

```

Figure 3.6: Rolled 2×2 base case

function call is a node of the tree), the base cases are the deepest level of the tree. Eliminating the deepest level of the octtree removes nearly *seven-eighths* of the nodes from the entire tree; in terms of function calls, raising the base case up one level removes seven-eighths of the function calls.

As levels of recursion are eliminated by raising the base case, some of the zero padding of the quadtree matrix can end up in the base cases on the east and south edges of the matrix. Special algorithms for handling those edges could be written at the cost of extra testing and more code. Alternatively, the zeros in those edge quadrants could participate in the base case algorithms. It is faster to just do the work rather than test to avoid the padding at such a deep level in the tree.

It also pays to ignore those zeros for the purpose of decorating the base case blocks for many of the same reasons. So, a base case block is considered dense if any element of the block is non-zero as is decorated appropriately; a base case block is decorated **zero** if and only if *all* of its elements are zero.

The new `upMult()` function is presented in Figure 3.7. The function `upBaseCase()` would be the unfolded and rolled base case as in Figure 3.6. In the actual code, this was not a function call; the base case was inlined to avoid the function-call overhead.

3.4.3 Decoration Driven Multiplication

To deal with the decorations (see Section 1.6.5), the dual multiplication algorithms `upMult()` and `dnMult()` are needed in two forms: one form that tests the decoration and another that does not. The testless form is triggered when both A and B have a **dense** decoration.

```

upMult (QTMatrix C, QTMatrix A, QTMatrix B) {
  if (basecase (C))
    upBaseCase (C, A, B);
  else {
    upMult (nw(C), nw(A), nw(B));
    dnMult (nw(C), ne(A), sw(B));
    upMult (sw(C), se(A), sw(B));
    dnMult (sw(C), sw(A), nw(B));
    dnMult (se(C), sw(A), ne(B));
    upMult (se(C), se(A), se(B));
    dnMult (ne(C), ne(A), se(B));
    upMult (ne(C), nw(A), ne(B));
  }
}

```

Figure 3.7: Raised base case version of multiplication

Matrix addition is needed when either A or B is an identity matrix and the multiplication is cumulative (e.g., $(C += IB) \equiv (C += B)$). If the multiplication is destructive (e.g., $C = IB = B$), a simple matrix copy is needed. If A or B is zero, the multiplication can be ignored if the multiplication is cumulative; a function to zero out C is needed if the multiplication is destructive. Whether it is a copy or an addition, it reduces the complexity of the operation from $O(n^3)$ to $O(n^2)$.

The algorithms presented in this chapter test for the base case; they would be used on **dense**-decorated matrices. Decoration-testing versions are made by replacing the base case test with tests for the **zero**, **identity**, and **dense** decorations.

3.4.4 Destructive Multiplication

When multiplication is destructive (i.e., $C = AB$ as opposed to $C += AB$), first-visit versions of `upMult()` and `dnMult()` are used to avoid a separate traversal to zero

```

upMultFirst (QTMatrix C, QTMatrix A, QTMatrix B) {
  if (basecase (C))
    upBaseCaseFirst (C, A, B);
  else {
    upMultFirst (nw(C), nw(A), nw(B));
    dnMult      (nw(C), ne(A), sw(B));
    upMultFirst (sw(C), se(A), sw(B));
    dnMult      (sw(C), sw(A), nw(B));
    dnMultFirst (se(C), sw(A), ne(B));
    upMult      (se(C), se(A), se(B));
    dnMultFirst (ne(C), ne(A), se(B));
    upMult      (ne(C), nw(A), ne(B));
  }
}

```

Figure 3.8: First-visit version of multiplication

out C . In the first-visit versions, the first visit of a quadrant of C initializes it while the second visit can use the original accumulating version. The first-visit `upMult()` is a variation of the code in Figure 3.7 and is presented in Figure 3.8. The base case of this function calls a special base case, `upBaseCaseFirst()`, that overwrites C with AB . This could be as simple as zeroing out C and then calling `upBaseCase()` on the three matrices.

This first-visit function could be used to conform more to the behavior of `dgemm()` which actually computes $C = \alpha AB + \beta C$ where α and β are constants. The base case `upBaseCaseFirst()` would have to multiply C by β before adding αAB . The regular base case would not have to worry about βC at all.

```

upMultLast (QTMatrix C, QTMatrix A, QTMatrix B) {
  if (basecase (C)) {
    upBaseCase (C, A, B);
    decorateBaseCase (C);
  }
  else {
    upMult      (nw (C), nw (A), nw (B));
    dnMultLast  (nw (C), ne (A), sw (B));
    upMult      (sw (C), se (A), sw (B));
    dnMultLast  (sw (C), sw (A), nw (B));
    dnMult      (se (C), sw (A), ne (B));
    upMultLast  (se (C), se (A), se (B));
    dnMult      (ne (C), ne (A), se (B));
    upMultLast  (ne (C), nw (A), ne (B));
    decorateCurrentLevel (C);
  }
}

```

Figure 3.9: Last-visit version of multiplication

3.4.5 Decorating C

Last-visit versions of the algorithms are used to decorate C , called the last time each quadrant of C is visited. The last-visit `upMult()` is presented in Figure 3.9. The base case of this function can use the regular base case and decorate C afterward. In the recursive case, `upMultLast()` calls the last-visit functions the second (and last) time each quadrant of C is visited. After all quadrants are visited, the current level of the matrix is decorated using only the decorations of its quadrants; no recursion is required.

```

topUpMult (QTMatrix C, QTMatrix A, QTMatrix B) {
  if (basecase (C)) {
    upBaseCase (C, A, B);
    decorateBaseCase (C);
  }
  else {
    upMultFirst (nw(C), nw(A), nw(B));
    dnMultLast (nw(C), ne(A), sw(B));
    upMultFirst (sw(C), se(A), sw(B));
    dnMultLast (sw(C), sw(A), nw(B));
    dnMultFirst (se(C), sw(A), ne(B));
    upMultLast (se(C), se(A), se(B));
    dnMultFirst (ne(C), ne(A), se(B));
    upMultLast (ne(C), nw(A), ne(B));
    decorateCurrentLevel (C);
  }
}

```

Figure 3.10: Top-level multiplication function

3.4.6 Top Level Function

The first and last versions of the algorithms must be called from a top-level function like the one in Figure 3.10. This function incorporates the operations of both the first- and last-visit functions.

3.5 Experimental Results

The machines used to run these tests are described in Section 2.3. As mentioned in Section 2.4.1, each machine had its own manufacturer-provided optimized version of BLAS, so the `dgemm()` tested on each machine was tuned for that machine. The quadtree code was identical on all machines.

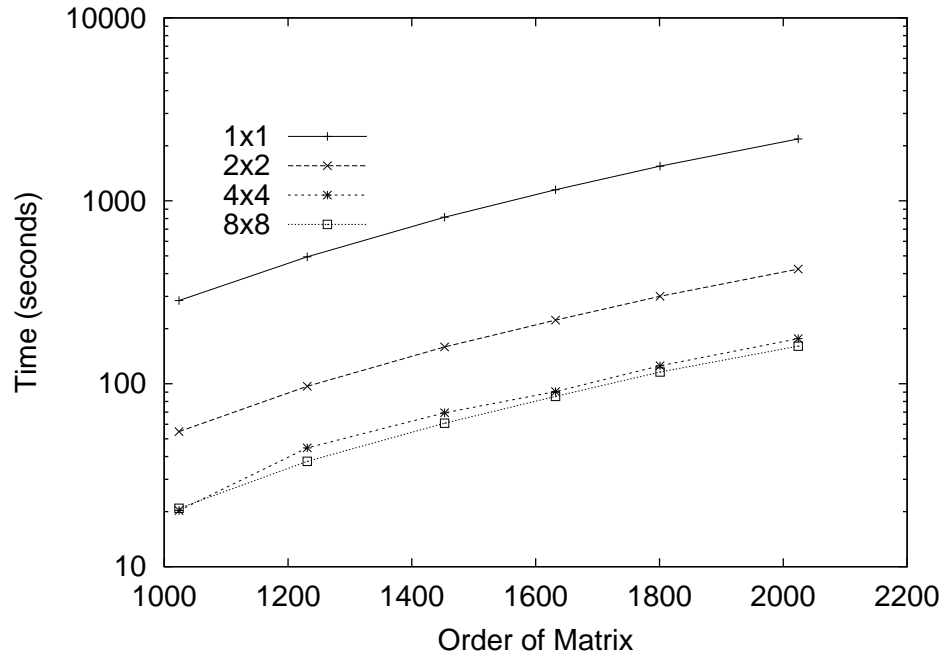


Figure 3.11: Running time of different base case sizes

3.5.1 Base Case Size

Figure 3.11 shows the running times on the Power Challenge for quadtree multiplication with different sizes for the base case. The improvements from unfolding the base cases are extremely significant; an 8×8 base case is on average more than 92% better than a 1×1 base case. The code reported on in earlier reports [20] was only unfolded to a two by two base case, a fact ignored by others [12]. The discovery of dilated integers [47] make unfolding easier and more effective since the code could be written in the language of loops that the C optimizer could handle.

3.5.2 Flop Counts

The flop count for both matrix-multiplication algorithms was approximated at $2n^3$ where n is the order of the matrix. This approximation was used in computing the millions of flops per second (mflop/s).

3.5.3 Quadtree Algorithm Versus BLAS Algorithm

The running times and mflop/s for `dgemm()` and quadtree multiplication are graphed in Figures 3.12 through 3.19. On the Power Challenge and the two Suns, the quadtree algorithm performs steadily although `dgemm()` consistently beats it except on the Octane.

The performance on the Power Challenge (Figures 3.12 and 3.13) for both algorithms is very consistent. The `dgemm()` function is very consistent at nearly two-thirds of the maximum flops, while the quadtree algorithm is just under half of the maximum flops. The penalty incurred by the padding in the quadtree algorithm can be seen at orders just past a power of two. For example, the mflop/s at order 2048 is 132.7 mflop/s, and 127.0 mflop/s at order 2094. The performance hit is two-fold: extra flops are incurred processing padding elements in the base cases, plus extra time is needed to process the decorations. Yet, the penalty is not that great: the mflop/s at order 2094 is still 95% of the mflop/s of order 2048.

The results on the Octane (Figures 3.14 and 3.15) are the most interesting. Striding (Section 2.4.2) appears to be an issue at order 2048 (see the results for *QR* factorization in Section 5.7.2). The more interesting results on the Octane are for

orders greater than 2500 where the performance of `dgemm()` falls apart. The manufacturer's `dgemm()` fails convincingly because it cannot handle the memory transfers in demand paging. That is, that code apparently was not designed for problems this big on RAM so small, and so its time explodes at orders greater than 2500. Figure 3.14 includes a plot of the number of major page faults triggered by `dgemm()` on the Octane. From that graph the virtual-memory problem is obvious; in contrast, the plot of the quadtree algorithm is smooth throughout the entire range.

This is *not* to suggest that on this machine BLAS should be used on small matrices and the quadtree algorithm on large matrices—this is *not* the quadtree matrix niche. BLAS can be fixed: change `dgemm()` by adding another level of tiling using specific knowledge about the virtual memory system. The important lesson here is that the quadtree algorithm takes this extra level of the memory hierarchy without extra or special effort. *The quadtree algorithm is implicitly tuned for all levels of the hierarchy without any knowledge about the hierarchy.*

The performance of the quadtree algorithm on both of the Suns (Figures 3.16 through 3.19) is dead smooth. Each new layer of the memory hierarchy is handled without changing the code. The performance of `dgemm()` is a bit more erratic, particularly at orders that are large powers of two, like 2048, 4096, and 8192. Just as on the Octane, striding is most likely the culprit for the bad performance at these orders.

Overall, these graphs suggest that striding is not an issue for quadtree matrices. They also suggest that the quadtree algorithm is already tuned for the all levels of the hierarchy without being re-tuned for each machine with knowledge of machine particulars. The performance of `dgemm()`, which *is* re-tuned for each machine with

knowledge of machine particulars, is usually better unless it was not tuned for a particular level of the hierarchy like on the Octane. The problem with the quadtree algorithm does not appear to be due to memory use; other results [48] suggest that the problem is the poor compilation of recursion.

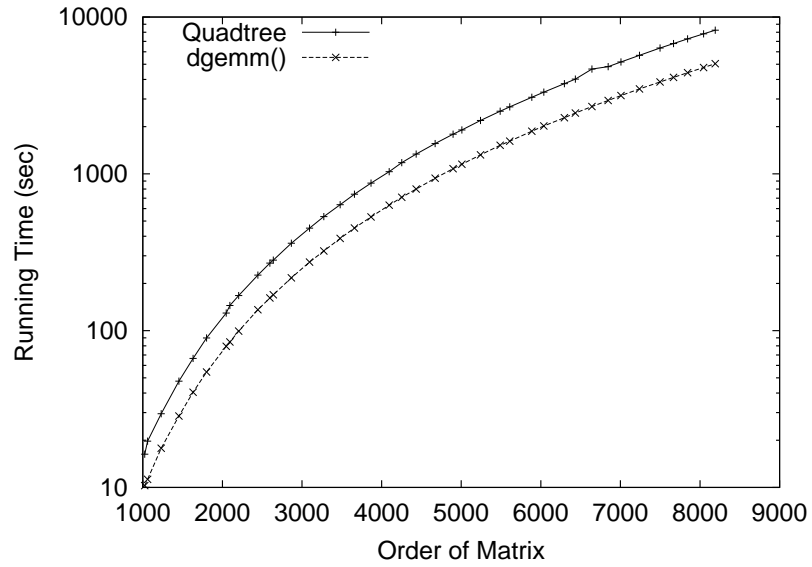


Figure 3.12: Running time of uniprocessor multiplication on Power Challenge

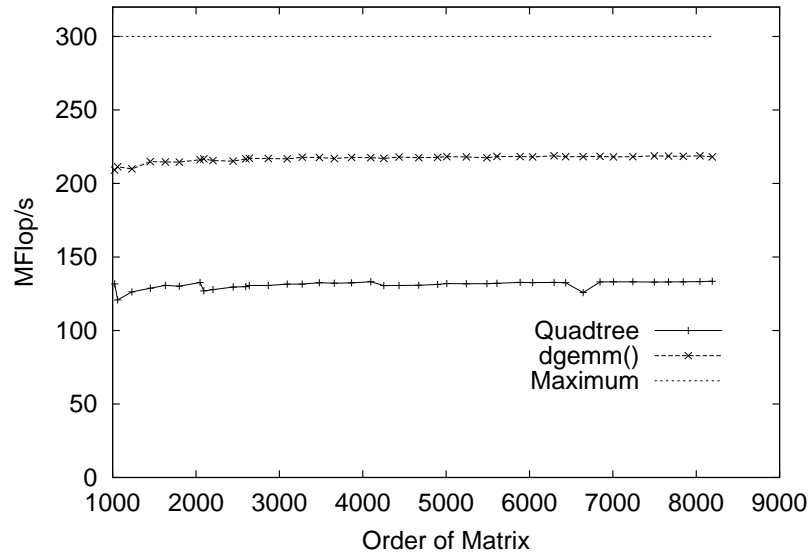


Figure 3.13: Mflop/s of uniprocessor multiplication on Power Challenge

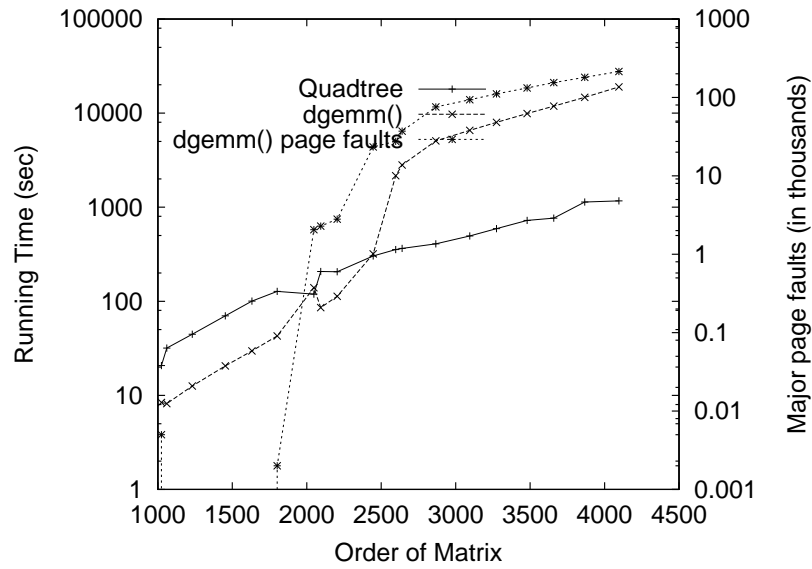


Figure 3.14: Running time of uniprocessor multiplication on Octane

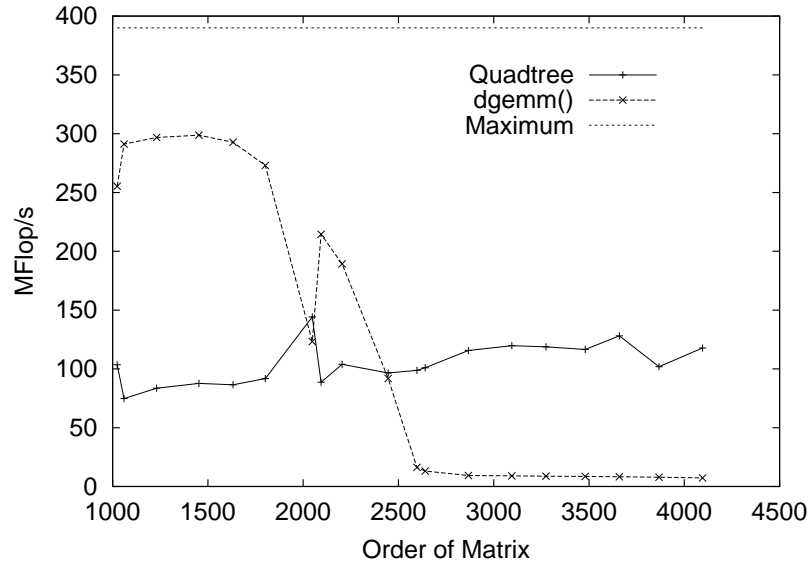


Figure 3.15: Mflop/s of uniprocessor multiplication on Octane

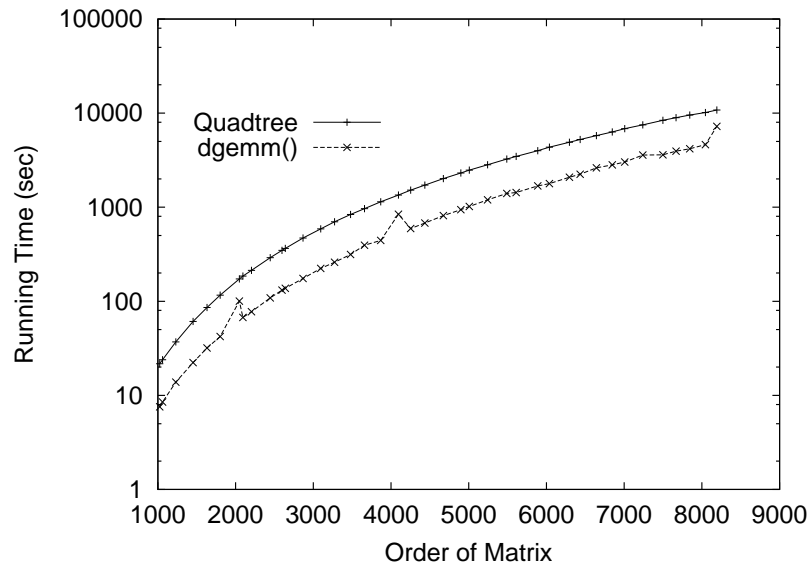


Figure 3.16: Running time of uniprocessor multiplication on Enterprise 450

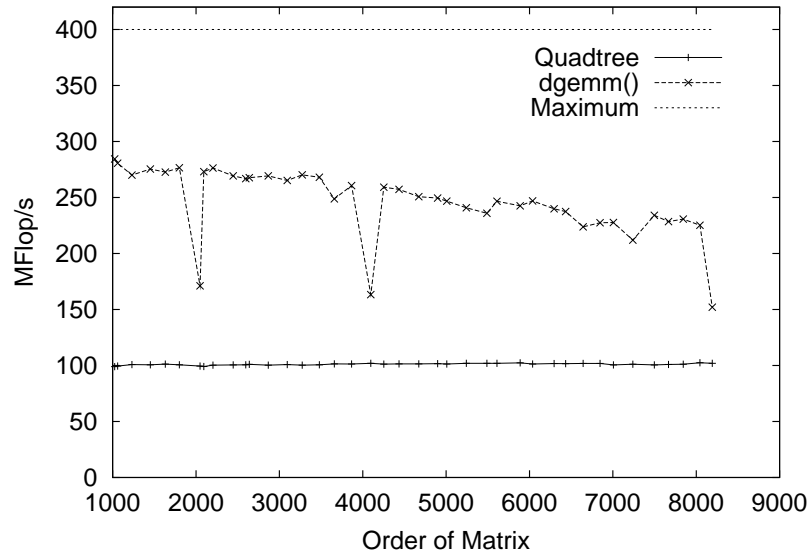


Figure 3.17: Mflop/s of uniprocessor multiplication on Enterprise 450

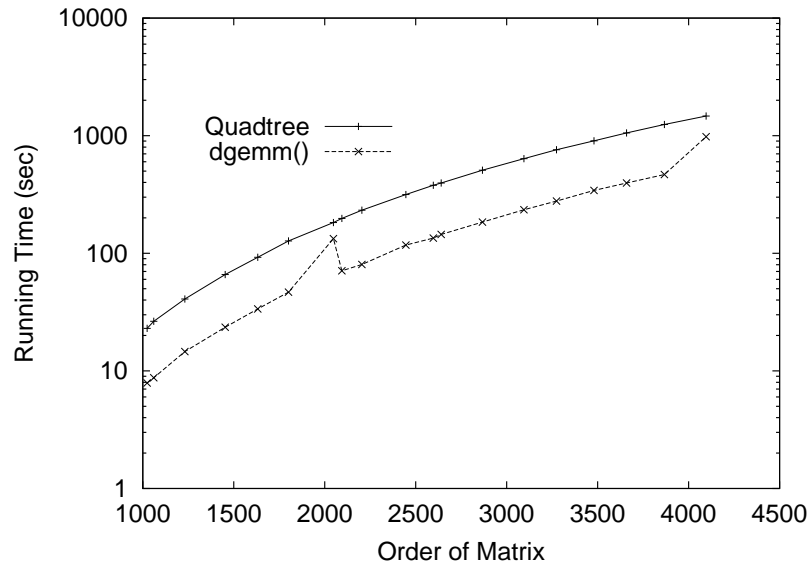


Figure 3.18: Running time of uniprocessor multiplication on Ultra 5/10

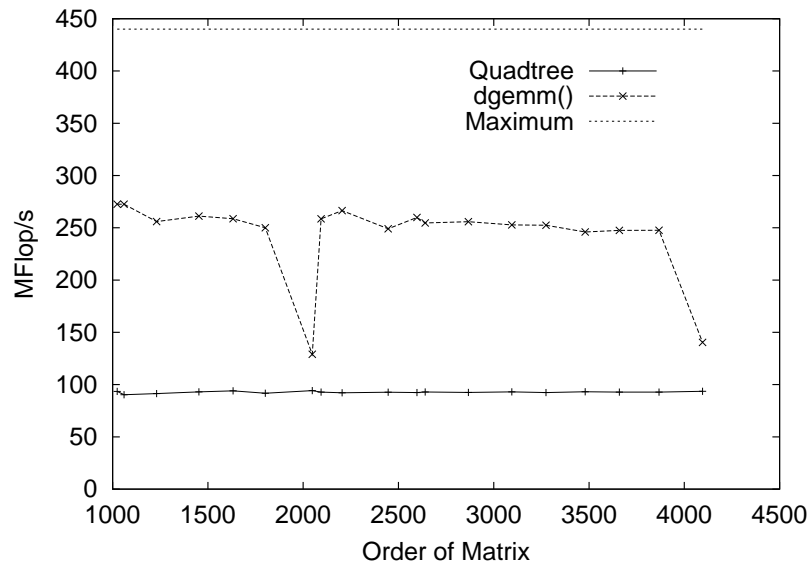


Figure 3.19: Mflop/s of uniprocessor multiplication on Ultra 5/10

4

Parallel Matrix Multiplication

The divide-and-conquer paradigm promises good parallelism, and the quadtree matrix algorithm delivers on this promise.

4.1 Dispatching Parallel Processes

The quadtree matrix multiplications of Figure 3.4 are divide-and-conquer algorithms; they divide matrix multiplication into eight recursive calls on the quadrants of the three matrices. As noted in Section 1.2.2, divide-and-conquer algorithms offer a simple way to parallelize the code: solve independent subproblems (i.e., execute independent function calls) in parallel. If there are more processors than independent problems, each parallel process can be given a proportion of the processors for further dispatches.

A programmer must first identify the independent function calls. For matrix multiplication, A and B can be read by two (or more) different processes at the same time without causing conflicts. However, only one process should write to any block of

C at a time. Consequently, multiplications involving the same quadrant of C should be done serially deferring parallelism one level down in the recursion.

It is also important to reduce the number of parallel dispatches to minimize the overhead of time and resources for process dispatch and synchronization [14, 28, 1]. If the execution of these functions are viewed as an octree of function calls, dispatches should occur at higher levels of the function-call tree. Yet, due to the zero padding, not all multiplications require the same amount of work. For balanced parallel processes and easy dispatch, it is best to dispatch two processes with the same amount of work involved. This observation suggests forcing parallel dispatch down to the deeper levels of the function-call tree where the subtrees (correspondingly, the parallel processes) are better balanced.

4.2 Eight Multiplication Patterns

There are eight different patterns that arise in the multiplication of square matrices based on the padding. These patterns are visually described and named in Figure 4.1 [20].

In all cases except for the perfect-square case, majority and minority padding (Definition 1.19) determine what recursive multiplications must be done and how the subprocesses can be balanced with each other. The pictures in Figure 4.1 display a mutually inconsistent variety of majority- and minority-padding possibilities.

In the following case analysis, the parallel dispatch for each case is described with a chart as in Figure 4.2. The charts are two dimensional diagrams: the vertical dimension is time; the horizontal dimension suggests parallel dispatch across available

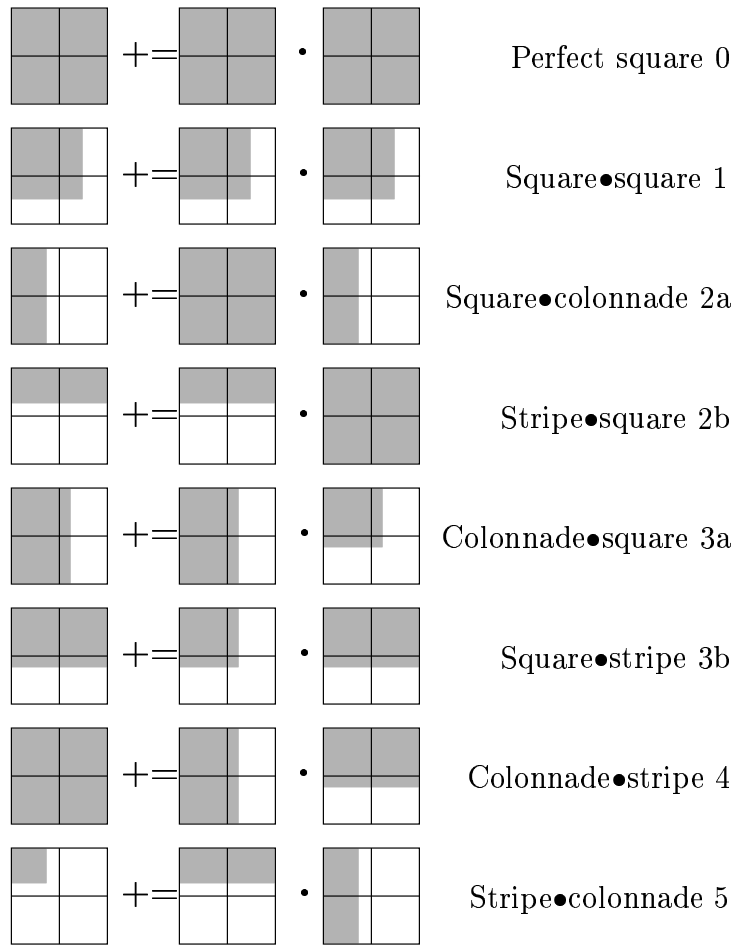


Figure 4.1: Parallel multiplication patterns

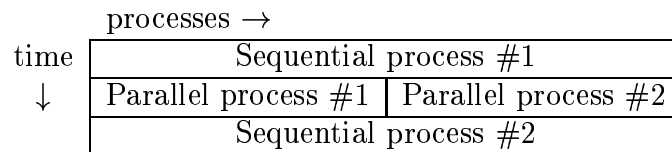


Figure 4.2: Parallel dispatch chart

processors. A horizontal bar represents synchronization. The steps between horizontal bars can be interleaved in any order. The steps are ordered by decreasing amount of work, but steps listed in parallel are balanced with respect to the work they do and their memory footprints.

For each case, parallel dispatch is done by splitting the work into two parallel processes to simplify process management. If there are more than two processors available, each parallel process is given half of the available processors for further dispatches. The parallel algorithm continues to dispatch deeper in the function-call tree. For a uniform distribution of processors, the number of processors is assumed to be a power of two.

0. Perfect Square. When A and B are perfect quadtree matrices (see Definition 1.19), the eight multiplications are all balanced, and so this case parallelizes very nicely. The eight recursive calls are split in half by splitting C west-east as in Figure 4.3. Alternatively, C could be split north-south or even diagonally (i.e., northwest and southeast versus northeast and southwest) for equally balanced processes.

If there are enough processors, the processor pool could be split four ways and the work dispatched as indicated in Figure 4.4. This four-way dispatch may be favored if many processors are available that should be taken advantage of early in the dispatch process. However, the two-way dispatch is still needed when there are fewer than four processors available. For simplicity, only the two-way dispatch is used.

The choice between two-way and four-way dispatch is possible because all of

the recursions are balanced in terms of processor and memory use. Other cases are constrained by the shapes and sizes of their padding and so do not have this same flexibility.

Each quadrant of C is affected by two of the recursions; and since the quadrants of C are written to, these two recursions must be synchronized; so they are assigned to the same parallel process, or they have an explicit synchronization step between them. Consequently, an eight-way dispatch is not possible for this (or any) case.

1. **Square•Square.** This is the general case, and this case serves as the root of the function-call tree.

MAJORITY-PADDING DISPATCH: Since all data is in the northwest quadrants of all three matrices, this case reduces to one recursive call to itself; parallelism is deferred to a lower level of the function-call tree.

MINORITY-PADDING DISPATCH: All eight cases arise from the eight multiplications. Since all of the stripes and colonnades of A and B have the same dimensions, the multiplications involving the northeast of C can be balanced against the multiplications involving the southwest of C ; the other multiplications do not balance with each other and are done serially, deferring the parallelism. This pattern of dispatch is given in Figure 4.5.

The last two steps in this case are well balanced in terms of computational complexity, but *not* in terms of their memory footprints. The colonnade•stripe case generates a perfect quadtree matrix C while the stripe•colonnade case results in a C with padding. This not only affects the memory accesses of the

$C \downarrow nw \ += \ A \downarrow nw B \downarrow nw$ perfect square	$C \downarrow se \ += \ A \downarrow sw B \downarrow ne$ perfect square
$C \downarrow nw \ += \ A \downarrow ne B \downarrow sw$ perfect square	$C \downarrow se \ += \ A \downarrow se B \downarrow se$ perfect square
$C \downarrow sw \ += \ A \downarrow se B \downarrow sw$ perfect square	$C \downarrow ne \ += \ A \downarrow ne B \downarrow se$ perfect square
$C \downarrow sw \ += \ A \downarrow sw B \downarrow nw$ perfect square	$C \downarrow ne \ += \ A \downarrow nw B \downarrow ne$ perfect square

Figure 4.3: Parallel multiplication: perfect square dispatch

$C \downarrow nw \ += \ A \downarrow nw B \downarrow nw$ perfect square	$C \downarrow se \ += \ A \downarrow sw B \downarrow ne$ perfect square	$C \downarrow sw \ += \ A \downarrow se B \downarrow sw$ perfect square	$C \downarrow ne \ += \ A \downarrow ne B \downarrow se$ perfect square
$C \downarrow nw \ += \ A \downarrow ne B \downarrow sw$ perfect square	$C \downarrow se \ += \ A \downarrow se B \downarrow se$ perfect square	$C \downarrow sw \ += \ A \downarrow sw B \downarrow nw$ perfect square	$C \downarrow ne \ += \ A \downarrow nw B \downarrow ne$ perfect square

Figure 4.4: Parallel multiplication: perfect square dispatch (4 processes)

$C \downarrow nw \ += \ A \downarrow nw B \downarrow nw$ perfect square $C \downarrow se \ += \ A \downarrow se B \downarrow se$ square•square	
$C \downarrow ne \ += \ A \downarrow nw B \downarrow ne$ square•colonnade $C \downarrow ne \ += \ A \downarrow ne B \downarrow se$ colonnade•square	$C \downarrow sw \ += \ A \downarrow sw B \downarrow nw$ stripe•square $C \downarrow sw \ += \ A \downarrow se B \downarrow sw$ square•stripe
$C \downarrow nw \ += \ A \downarrow ne B \downarrow sw$ colonnade•stripe $C \downarrow se \ += \ A \downarrow sw B \downarrow ne$ stripe•colonnade	

Figure 4.5: Parallel multiplication: square•square dispatch

two cases but also the possibilities for future parallel dispatches in the function-call tree.

2. The next two cases are grouped together since they are symmetric in their matrix patterns and are thus computationally equivalent both in terms of flops and their memory footprints. C is split in half to divide up the multiplications into two parallel processes.

- a. **Square•Colonnade.** C is split north-south to yield balanced processes. C could be split diagonally, but not east-west.

MAJORITY-PADDING DISPATCH: This case consists of four multiplications of this same case that are nicely balanced. This pattern of dispatch is given in Figure 4.6.

MINORITY-PADDING DISPATCH: This case consists of four multiplications of this same type and four perfect-square cases; the work is well balanced for good parallelism. This pattern of dispatch is given in Figure 4.7.

- b. **Stripe•Square.** C is split east-west to yield balanced processes. C could be split along its diagonals, but not north-south.

MAJORITY-PADDING DISPATCH: There are four recursive cases as indicated in Figure 4.8 that parallelize nicely.

MINORITY-PADDING DISPATCH: This case consists of four recursive cases as well as four perfect-square cases, and all parallelize nicely. This pattern of dispatch is given in Figure 4.9.

3. The next two cases are also symmetric and computationally equivalent.

$C \downarrow_{nw} += A \downarrow_{nw} B \downarrow_{nw}$ square•colonnade	$C \downarrow_{sw} += A \downarrow_{sw} B \downarrow_{nw}$ square•colonnade
$C \downarrow_{nw} += A \downarrow_{ne} B \downarrow_{sw}$ square•colonnade	$C \downarrow_{sw} += A \downarrow_{se} B \downarrow_{sw}$ square•colonnade

Figure 4.6: Parallel multiplication: square•colonnade majority-padding dispatch

$C \downarrow_{nw} += A \downarrow_{nw} B \downarrow_{nw}$ perfect square	$C \downarrow_{sw} += A \downarrow_{sw} B \downarrow_{nw}$ perfect square
$C \downarrow_{nw} += A \downarrow_{ne} B \downarrow_{sw}$ perfect square	$C \downarrow_{sw} += A \downarrow_{se} B \downarrow_{sw}$ perfect square
$C \downarrow_{ne} += A \downarrow_{ne} B \downarrow_{se}$ square•colonnade	$C \downarrow_{se} += A \downarrow_{se} B \downarrow_{se}$ square•colonnade
$C \downarrow_{ne} += A \downarrow_{nw} B \downarrow_{ne}$ square•colonnade	$C \downarrow_{se} += A \downarrow_{sw} B \downarrow_{ne}$ square•colonnade

Figure 4.7: Parallel multiplication: square•colonnade minority-padding dispatch

$C \downarrow_{nw} += A \downarrow_{nw} B \downarrow_{nw}$ stripe•square	$C \downarrow_{ne} += A \downarrow_{nw} B \downarrow_{ne}$ stripe•square
$C \downarrow_{nw} += A \downarrow_{ne} B \downarrow_{sw}$ stripe•square	$C \downarrow_{ne} += A \downarrow_{ne} B \downarrow_{se}$ stripe•square

Figure 4.8: Parallel multiplication: stripe•square majority-padding dispatch

$C \downarrow_{nw} += A \downarrow_{nw} B \downarrow_{nw}$ perfect square	$C \downarrow_{ne} += A \downarrow_{nw} B \downarrow_{ne}$ perfect square
$C \downarrow_{nw} += A \downarrow_{ne} B \downarrow_{sw}$ perfect square	$C \downarrow_{ne} += A \downarrow_{ne} B \downarrow_{se}$ perfect square
$C \downarrow_{sw} += A \downarrow_{se} B \downarrow_{sw}$ stripe•square	$C \downarrow_{se} += A \downarrow_{se} B \downarrow_{se}$ stripe•square
$C \downarrow_{sw} += A \downarrow_{sw} B \downarrow_{nw}$ stripe•square	$C \downarrow_{se} += A \downarrow_{sw} B \downarrow_{ne}$ stripe•square

Figure 4.9: Parallel multiplication: stripe•square minority-padding dispatch

- a. **Colonnade•Square.** C is partitioned north-south to yield balanced processes. C could be split diagonally, but not east-west.

MAJORITY-PADDING DISPATCH: This case reduces to two recursive cases that parallelize nicely. The pattern of dispatch is given in Figure 4.10.

MINORITY-PADDING DISPATCH: This case reduces to four recursive cases and a variety of other cases that balance and parallelize nicely. This pattern of dispatch is given in Figure 4.11.

- b. **Square•Stripe.** C is partitioned east-west to yield balanced processes. C could be split diagonally, but not north-south.

MAJORITY-PADDING DISPATCH: This case consists of two recursive calls that parallelize nicely. This pattern of dispatch is given in Figure 4.12.

MINORITY-PADDING DISPATCH: This consists of a variety of cases that parallelize nicely. This pattern of dispatch is given in Figure 4.13.

4. **Colonnade•Stripe.** This case can be viewed as an outer product of two vectors. C is split north-south to yield balanced processes; it can also be split east-west or diagonally for balanced processes.

MAJORITY-PADDING DISPATCH: This case consists of four recursive cases that parallelize nicely. This pattern of dispatch is given in Figure 4.14.

MINORITY-PADDING DISPATCH: This case consists of a variety of other cases that parallelize nicely. This pattern of dispatch is given in Figure 4.15.

5. **Stripe•Colonnade.** This case can be viewed as an inner product of vectors. Since the area of C that is updated is smaller than the area of A and B , much of this case must be done serially.

$C \downarrow nw \ += \ A \downarrow nwB \downarrow nw$ colonnade•square	$C \downarrow sw \ += \ A \downarrow swB \downarrow nw$ colonnade•square
-----------------------------------------------------------------------------	-----------------------------------------------------------------------------

Figure 4.10: Parallel multiplication: colonnade•square majority-padding dispatch

$C \downarrow nw \ += \ A \downarrow nwB \downarrow nw$ perfect square	$C \downarrow sw \ += \ A \downarrow swB \downarrow nw$ perfect square
$C \downarrow nw \ += \ A \downarrow neB \downarrow sw$ colonnade•stripe	$C \downarrow sw \ += \ A \downarrow seB \downarrow sw$ colonnade•stripe
$C \downarrow ne \ += \ A \downarrow neB \downarrow se$ colonnade•square	$C \downarrow se \ += \ A \downarrow seB \downarrow se$ colonnade•square
$C \downarrow ne \ += \ A \downarrow nwB \downarrow ne$ square•colonnade	$C \downarrow se \ += \ A \downarrow swB \downarrow ne$ square•colonnade

Figure 4.11: Parallel multiplication: colonnade•square minority-padding dispatch

$C \downarrow nw \ += \ A \downarrow nwB \downarrow nw$ square•stripe	$C \downarrow ne \ += \ A \downarrow nwB \downarrow ne$ square•stripe
--------------------------------------------------------------------------	--------------------------------------------------------------------------

Figure 4.12: Parallel multiplication: square•stripe majority-padding dispatch

$C \downarrow nw \ += \ A \downarrow nwB \downarrow nw$ perfect square	$C \downarrow ne \ += \ A \downarrow nwB \downarrow ne$ perfect square
$C \downarrow nw \ += \ A \downarrow neB \downarrow sw$ colonnade•stripe	$C \downarrow ne \ += \ A \downarrow neB \downarrow se$ colonnade•stripe
$C \downarrow sw \ += \ A \downarrow seB \downarrow sw$ square•stripe	$C \downarrow se \ += \ A \downarrow seB \downarrow se$ square•stripe
$C \downarrow sw \ += \ A \downarrow swB \downarrow nw$ stripe•square	$C \downarrow se \ += \ A \downarrow swB \downarrow ne$ stripe•square

Figure 4.13: Parallel multiplication: square•stripe minority-padding dispatch

$C \downarrow nw \ += \ A \downarrow nwB \downarrow nw$ colonnade•stripe	$C \downarrow ne \ += \ A \downarrow nwB \downarrow ne$ colonnade•stripe
$C \downarrow sw \ += \ A \downarrow swB \downarrow nw$ colonnade•stripe	$C \downarrow se \ += \ A \downarrow swB \downarrow ne$ colonnade•stripe

Figure 4.14: Parallel multiplication: colonnade•stripe majority-padding dispatch

$C \downarrow nw \ += \ A \downarrow nwB \downarrow nw$ perfect square	$C \downarrow ne \ += \ A \downarrow nwB \downarrow ne$ perfect square
$C \downarrow nw \ += \ A \downarrow neB \downarrow sw$ colonnade•stripe	$C \downarrow ne \ += \ A \downarrow neB \downarrow se$ colonnade•stripe
$C \downarrow sw \ += \ A \downarrow seB \downarrow sw$ colonnade•stripe	$C \downarrow se \ += \ A \downarrow seB \downarrow se$ colonnade•stripe
$C \downarrow sw \ += \ A \downarrow swB \downarrow nw$ perfect square	$C \downarrow se \ += \ A \downarrow swB \downarrow ne$ perfect square

Figure 4.15: Parallel multiplication: colonnade•stripe minority-padding dispatch

$C \downarrow \text{nw} += A \downarrow \text{nw} B \downarrow \text{nw}$ <small>stripe•colonnade</small>
$C \downarrow \text{nw} += A \downarrow \text{ne} B \downarrow \text{sw}$ <small>stripe•colonnade</small>

Figure 4.16: Parallel multiplication: stripe•colonnade majority-padding dispatch

$C \downarrow \text{nw} += A \downarrow \text{nw} B \downarrow \text{nw}$ <small>perfect square</small>	
$C \downarrow \text{nw} += A \downarrow \text{ne} B \downarrow \text{sw}$ <small>perfect square</small>	
$C \downarrow \text{ne} += A \downarrow \text{ne} B \downarrow \text{se}$ <small>square•colonnade</small>	$C \downarrow \text{sw} += A \downarrow \text{se} B \downarrow \text{sw}$ <small>stripe•square</small>
$C \downarrow \text{ne} += A \downarrow \text{nw} B \downarrow \text{ne}$ <small>square•colonnade</small>	$C \downarrow \text{sw} += A \downarrow \text{sw} B \downarrow \text{nw}$ <small>stripe•square</small>
$C \downarrow \text{se} += A \downarrow \text{sw} B \downarrow \text{ne}$ <small>stripe•colonnade</small>	
$C \downarrow \text{se} += A \downarrow \text{se} B \downarrow \text{se}$ <small>stripe•colonnade</small>	

Figure 4.17: Parallel multiplication: stripe•colonnade minority-padding dispatch

MAJORITY-PADDING DISPATCH: There are two recursive calls to this same case that must be done serially since both update $C \downarrow \text{nw}$. This pattern of dispatch is given in Figure 4.16.

MINORITY-PADDING DISPATCH: Similar to the square•square case, the multiplications for the northeast of C are well balanced against the multiplications for the southwest of C for some parallelism; the other multiplications must be done serially. This pattern of dispatch is given in Figure 4.17.

4.3 Implementation and Compiler Issues

Section 2.4 discusses the basic compiler issues. There are also some particular considerations for parallel matrix multiplication.

4.3.1 Padding Size

Scheduling around the zero padding has an overhead; at some point it is best to consider the matrix to be perfect square and dispatch accordingly. For example, it seems fairly silly to treat a 511×511 matrix as anything but dense 512×512 . The extra row and column of padding would hardly make a difference in the parallel dispatch, especially with an unfolded base case (see Section 3.4.2). Experiments on the Power Challenge indicate that a padding of up to order 32 is worth ignoring. This same size was used on the Enterprise 450.

4.3.2 Uniprocessor-Commit Size

Due to the overhead of a dispatch [14, 28, 1] it may be faster to process a small matrix with one processor even if there are multiple processors available.

Definition 4.1 *The uniprocessor-commit size is the largest size of a quadtree matrix where the overhead of parallel dispatch and the time to execute the parallel processes is greater than executing the uniprocessor algorithm.*

The base case must be done uniprocessor, so the uniprocessor-commit size for multiplication must be at least as large as the base case size (see Section 3.4.2).

Experiments on the Power Challenge indicate that the padding size is perhaps more important than the uniprocessor-commit size: all uniprocessor-commit sizes smaller than the padding size work equally well; larger sizes degrade performance since some parallelism is lost. So the uniprocessor-commit size was set to the base case size on both parallel machines.

4.4 Experimental Results

The machines used to run these tests are described in Section 2.3. Only the Power Challenge and the Sun Enterprise machines are parallel machines, so only they were used to run the parallel experiments. Speed-ups were calculated as described in Section 2.5.1.

The parallel speed-ups for the quadtree multiplication and `dgemm()` on the Power Challenge are graphed in Figures 4.18 and 4.19. Both algorithms do the same amount of work, and the speed-ups are fairly comparable. The speed-ups of the quadtree algorithm on the Power Challenge (Figure 4.18) are near the linear ideals for two and four processors. The super-linear speed-ups at order 6643 are inaccurate since the uniprocessor time at that order is a little higher than it should be (see Figure 3.12). Not surprisingly, the parallelism seems to be best at power-of-two orders (like orders 2048 and 4096) and improves as the matrix gets larger. Power-of-two orders have the best parallelism since every parallel dispatch is a well balanced, perfect-square dispatch. Larger matrices perform better because there is more work done for each dispatch, amortizing the cost of the overhead of the parallel dispatch.

Eight processors on the Power Challenge starts off rather low where the payoff for parallel dispatch is not very good—too many dispatches are done deep in the function-call tree. Again, power-of-two orders have quite good speed-ups since all of the dispatches are perfect-square dispatches. But, as the order of the matrix increases, the overall performance at eight processors steadily increases to the ideal since there is a better payoff for the parallel dispatches.

The speed-up for `dgemm()` on the Power Challenge (Figure 4.19) also suffers problems at smaller orders, but it does better as the order increases. The performance rapidly approaches the ideal as asymptote.

The speed-ups for the Enterprise 450 are graphed in Figures 4.20 and 4.21. The performance of the quadtree algorithm (Figure 4.20) is very impressive. At two processors, the speed-up is consistently just under the ideal; at four processors, the speed-up is a little inconsistent for small orders, but increases nicely and never dips below 3.5. The performance of `dgemm()` (Figure 4.21) appears to be a little more ragged, especially for four processors. The speed-ups also appear to be super linear most of the time for two processors. Both the raggedness and the super-linear speed-ups are due to using wall-clock time for computing the speed-up (see Section 2.5.2). Most likely, the speed-up is right around the ideals.

Overall, the parallel speed-up of the quadtree algorithm is quite good. The algorithm pays some penalties when it must deal with non-power-of-two orders, but this is only a real problem on the Power Challenge with eight processors (Figure 4.18). But it is very competitive with parallel `dgemm()`. This all strongly suggests that the divide-and-conquer paradigm use for parallelism can be successful.

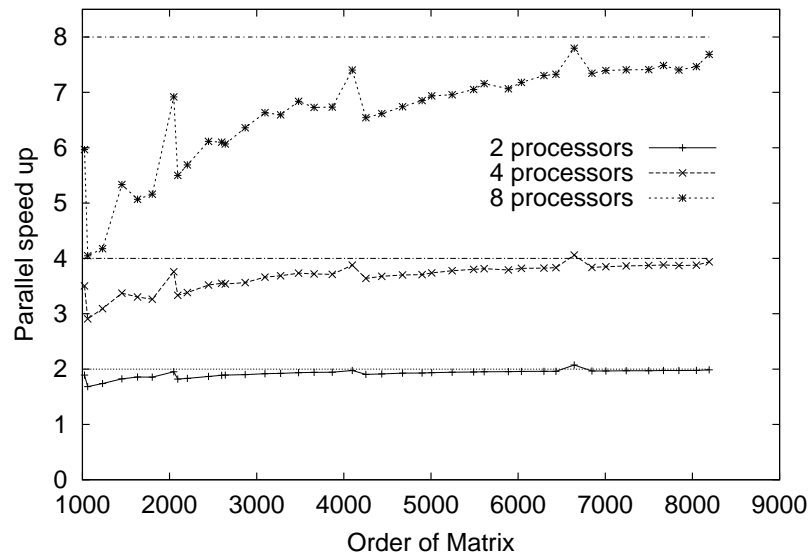


Figure 4.18: Speed-up for quadtree multiplication on Power Challenge

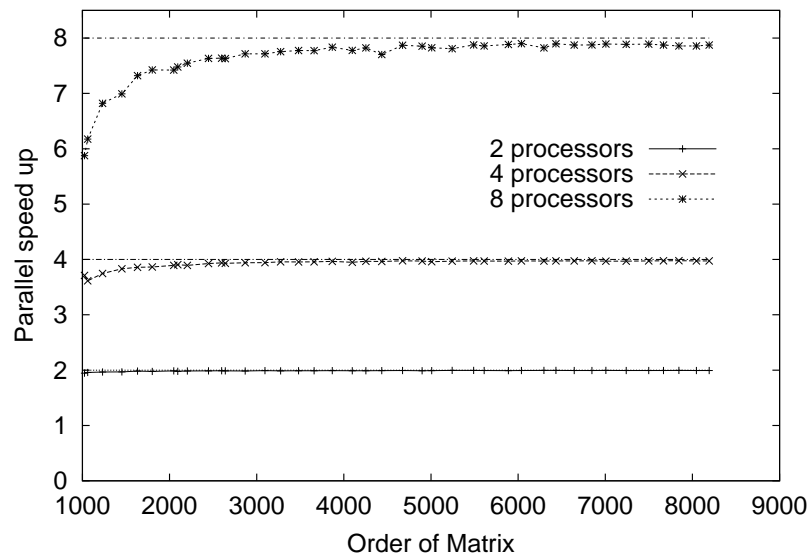


Figure 4.19: Speed-up for `dgemm()` on Power Challenge

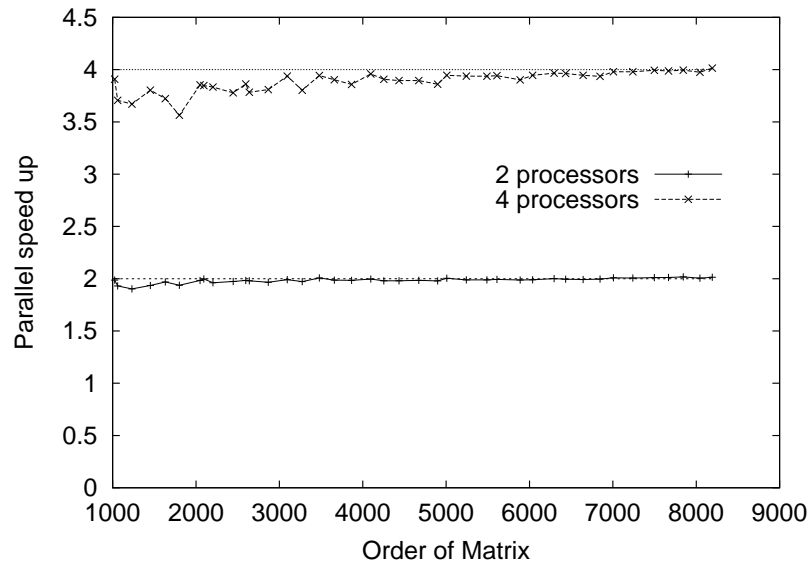


Figure 4.20: Speed-up for quadtree multiplication on Enterprise 450

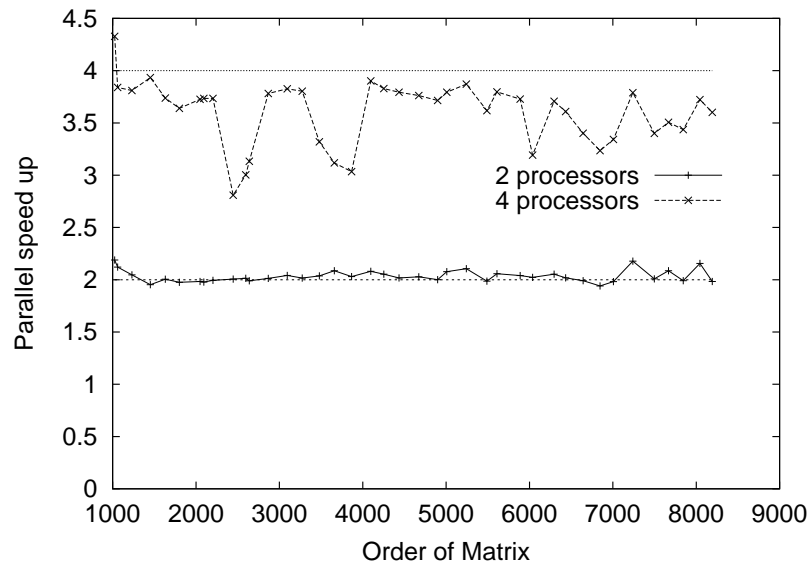


Figure 4.21: Speed-up for dgemm() on Enterprise 450

5

QR Factorization

As discussed in Section 1.4.3, a *QR* factorization factors a matrix A into an orthogonal matrix Q and an upper-triangular matrix R . This factorization is useful for solving the least squares problem and for finding eigenvalues. This chapter explores a quadtree algorithm for *QR* factorization.

5.1 The Basic Algorithms

The *QR* factorization of an $n \times n$ matrix A produces an $n \times n$ orthogonal matrix Q and an $n \times n$ upper-triangular matrix R such that $A = QR$ [24, Section 5.2]. A *QR* factorization applies a series of orthogonal transformations $Q_1, Q_1, Q_2, \dots, Q_m$ to $A = A_0, A_1, A_2, \dots$, updating each successive $A_i = Q_i^T A_{i-1}$ until $A_m = R$ is produced. (The value m depends on the particular transformation used.) The matrix Q is formed by multiplying the individual orthogonal transformations together. Different *QR* factorizations arise because of the different orthogonal transformations that can be used for Q_i .

5.1.1 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 [24, 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 [24, Section 5.2.1].

5.1.2 Givens QR Factorization

A *Givens rotation* eliminates just one selected element from the matrix using another element [24, Section 5.1.8]. To eliminate $b \neq 0$ using a , the Givens rotation is formed from the cosine c and sine s of an a - b right triangle:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} \quad (5.1)$$

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. Golub and Van Loan [24, Section 5.2.3] present a QR factorization algorithm using Givens rotations; a version of this algorithm is presented in Figure 5.1 in a loose C syntax.

```

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 5.1: Iterative, column-based QR factorization using Givens rotations

5.2 The Quadtree QR Factorization Algorithms

The algorithm for QR factorization for quadtree matrices is done with two mutually recursive functions, f and e , that solve two different, but interrelated, problems.

5.2.1 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 $A = QR$. The function is presented in Figure 5.2.

If A is scalar (i.e., $n = 1$), then $Q = I = [1]$ and $R = A$. The recursive case breaks down into several steps: first, the northwest and southwest quadrants of A are factored recursively using f (Steps 1 and 2). Second, the modified southwest quadrant is eliminated with the elimination function e (see Section 5.2.2) using the modified northwest quadrant (Step 3). The Q s resulting from the first steps are multiplied together (Step 4), and the product is used to update the eastern colonnade of A (Step 5). The updated southeast of A is factored recursively (Step 6). Finally, the Q from the last factorization is multiplied with the previous Q s (Step 7).

5.2.2 Quadtree QR Eliminate

The elimination function $e: \langle N, S \rangle \mapsto \langle Q, \tilde{N} \rangle$ 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.) The

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: $\langle Q_1, R_1 \rangle = f(A \downarrow \mathbf{nw})$.

Step 2: $\langle Q_2, R_2 \rangle = f(A \downarrow \mathbf{sw})$.

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: $\begin{bmatrix} U_n \\ U_s \end{bmatrix} = Q_4^T \begin{bmatrix} A \downarrow \mathbf{ne} \\ A \downarrow \mathbf{se} \end{bmatrix}$.

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 5.2: QR factorization function, f

matrices 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 this function is that

$$\begin{bmatrix} N \\ S \end{bmatrix} = Q \begin{bmatrix} \tilde{N} \\ Z \end{bmatrix}. \quad (5.2)$$

The algorithm is presented in Figure 5.3.

If N and S are scalars, Q is the Givens rotation to eliminate S using N using Equation 5.1. \tilde{N} is the result of multiplying Q^T by N and S . In the recursive case, the function e is called on the northwest quadrants of N and S (Step 1). Then e is called on the southeast quadrants (Step 2). The Q from the first elimination is used to update the northeast quadrants of N and S (Step 3). This leaves only a square block in the northeast of S . It is first factored using f (Step 4), then it is eliminated using e and the southeast of N (Step 6). The last Q s are multiplied together (Step 7).

The elimination function presents some specialized multiplications that require fewer than $2n^3$ flops. Since N and S are both triangular, Q comes out of e with triangular patterns in it. (The Q from f is dense as one would expect.) The 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.

5.3 Tuning for the Memory Hierarchy

QR factorization with Householder reflections can be put into a block form [24, Section 5.2.2] for row- and column-major storage. Columns of A are factored as before,

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 5.2 holds.

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

Recursion when $n > 1$,

$$\text{Step 1: } \langle Q_1, \tilde{N}_1 \rangle = e(N \downarrow \text{nw}, S \downarrow \text{nw}).$$

$$\text{Step 2: } \langle Q_2, \tilde{N}_2 \rangle = e(N \downarrow \text{se}, S \downarrow \text{se}).$$

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

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

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

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

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

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

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

Figure 5.3: QR elimination function, e

but updates are only applied to the current colonnade block. When a new colonnade is started, all pending updates are applied before factoring the new colonnade. Not only does this avoid accessing these columns repeatedly for updates, but the block update is also a more efficient matrix-matrix operation.

Similar blocking is *naturally* part of the quadtree matrix algorithm; just like the quadtree matrix algorithm for matrix multiplication, blocking is automatic. For the QR factorization algorithms, updates to eastern colonnades are saved until the western colonnades are fully processed. Unlike quadtree-matrix multiplication, the QR factorization algorithm for the quadtree matrix does not reuse blocks, so there is no need to create dual versions of the algorithm for block reuse.

5.4 Successful QR Factorization

In addition to algebraic correctness, it is also necessary for the algorithms to avoid problematic operations and to avoid generating large errors.

5.4.1 Computational Correctness

The functions f and e work correctly:

Theorem 5.1 *This theorem consists of two clauses:*

Clause F *If $f: A_f \mapsto \langle Q_f, R_f \rangle$ is defined as in Figure 5.2 and A_f is an $n \times n$ matrix, then Q_f is an $n \times n$ orthogonal matrix, R_f is an $n \times n$ upper-triangular matrix, and $A_f = Q_f R_f$.*

Clause E If $e: \langle N_e, S_e \rangle \mapsto \langle Q_e, \tilde{N}_e \rangle$ is defined as in Figure 5.3 and both N_e and S_e are $n \times n$ upper-triangular matrices, then Q_e is a $2n \times 2n$ orthogonal matrix, \tilde{N}_e is an $n \times n$ upper-triangular matrix, and

$$\begin{bmatrix} N_e \\ S_e \end{bmatrix} = Q_e \begin{bmatrix} \tilde{N}_e \\ Z \end{bmatrix}.$$

Section A.3 proves this theorem.

It is also important that if A is non-singular, then Q and R are also non-singular:

Theorem 5.2 *Let A be an $n \times n$ matrix. Let $\langle Q, R \rangle = f(A)$ as defined in Figure 5.2. Then R is singular if and only if A is singular; Q is always non-singular.*

Section A.4 proves this theorem.

5.4.2 Avoiding Problematic Operations

QR factorization never divides by zero. The only division in the algorithms is in computing c and s of a Givens rotation, and the denominator of the divisions are zero only when $a = b = 0$. However, when $b = 0$ (regardless of a), the identity matrix is used as the rotation. So division by zero is avoided.

In computing a Givens rotation, the square root is always safe because a and b are real (not complex), non-zero numbers, and so $a^2 + b^2$ is always positive.

5.4.3 Error Analysis

Naive LU factorization can result in large errors [26, Chapter 9]. To avoid these errors, it is usually implemented with partial pivoting [24, Section 3.4.3]. The algorithm searches for large values in one step of the algorithm, and rows are exchanged so that the factorization is kept as accurate as possible [26, Chapter 9]. This pivoting also avoids division by zero.

In QR factorization, the orthogonality of Q gives it great stability. The error analysis of QR factorization using Givens rotations without pivoting is very favorable [26, Section 1.14.2, Section 18.5]. Pivoting is used with QR factorizations on singular matrices for reasons beyond the scope of this work. [24, Section 5.4.1] Pivoting is unnecessary on nonsingular matrices.

This only makes *explicit* pivoting unnecessary. If $N = Z$ is e , then the natural Givens rotation generated by e will, in fact, implicitly pivot the matrices so that $\tilde{N} = S$.

5.5 Implementation and Compiler Issues

See also the compiler issues of Section 2.4 and 3.4.

5.5.1 In-place Algorithms

The functions f and e as presented in Figures 5.2 and 5.3 (respectively) are purely functional and do not run in-place. Implementing them to run in-place can be done

by gradually replacing A by R while assembling Q in another matrix. Matrix multiplications of the forms $X = XY$ and $Y = XY$ are very useful for accumulating Q and updating A , but they require extra memory to accumulate partial sums. Thus, a temporary matrix is passed to each algorithm for storing intermediate results.

5.5.2 The Transpose of Q

The algorithms for the functions f and e actually return Q^T , rather than Q , because it is easier to update A using Q^T . It is easy to transpose a matrix after the factorization is complete.

5.5.3 Base Case Unfolding

Like matrix multiplication, the base case of QR factorization is unfolded in practice, here to an 8×8 block (see Section 3.4.2). The 8×8 block is factored using the loops in Figure 5.1. The loops of that code were translated for use with level-order indices as suggested by Wise [47].

5.5.4 Decoration Driven QR Factorization

No special cases for handling identity matrices are necessary in e or f because they rarely arise in A , originally or during the factorization. If there is an identity matrix in A , it will be factored correctly (i.e., $Q = R = I$); it will just be treated as a dense matrix. The identity matrices in Q are never factored, just multiplied, and the multiplication algorithm does treat them as special cases (see Section 3.4.3).

Zero matrices lead to some simplified work in e and f resulting mostly in identity transformations: $f: Z \mapsto \langle I, Z \rangle$ and $e: \langle N, Z \rangle \mapsto \langle I, N \rangle$. In both functions, the matrix to be factored or eliminated is already eliminated, so no changes must be made to A or N , and Q is trivially the identity matrix.

Due to the way C is visited by matrix multiplication, special first- and last-visit functions are needed to initialize and decorate C properly and efficiently (see Sections 3.4.4 and 3.4.5). There is no special matrix to initialize in f or e , so special first-visit functions are unnecessary. Last-visit functions are also unnecessary since Q , R , and \tilde{N} must always be decorated at the end of both functions.

5.5.5 Accumulating Q

Many algorithms for QR factorization do not accumulate Q explicitly. Usually, each Q_i is stored in the memory for the eliminated portion of A . However, to take advantage of the blocked efficiency of matrix multiplication, Q must be accumulated explicitly for the quadtree matrix algorithm. So Q is accumulated explicitly.

5.6 LAPACK

LAPACK (see Section 1.8) provides the function `dgeqrf()` for QR factorization (without pivoting). This function does not create a matrix Q , but stores the Householder reflections in the zeroed portions of A as it becomes R .¹ So, this function performs fewer flops than the quadtree matrix algorithm which accumulates Q explicitly.

¹ Q^T can be formed from this representation if necessary.

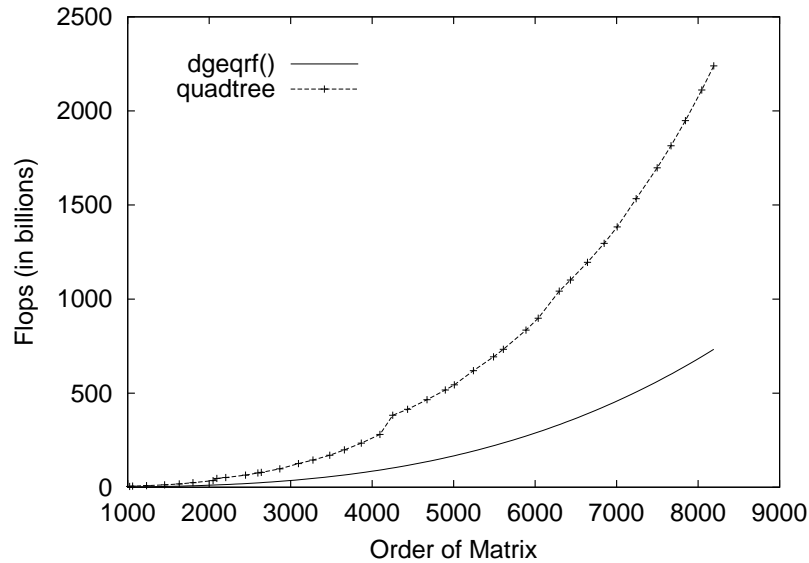


Figure 5.4: Flop count of QR factorization algorithms

The LAPACK library was compiled from scratch on the SGI machines, but linked to the hand-tuned BLAS library. The Sun Performance Library provides a hand-tuned LAPACK, and this was used on the Sun machines.

5.7 Experimental Results

5.7.1 Flop Count

The flop counts for `dgeqrf()` and the quadtree algorithm are graphed in Figure 5.4. The flop count for the quadtree matrix is higher because it accumulates Q explicitly, while `dgeqrf()` stores just the Household vectors (see Sections 5.5.5 and 5.6).

The flop counts for `dgeqrf()` were approximated at $4n^3/3$ flops [24, Section 5.2.1]; the actual number of flops may be slightly higher since the block Householder algorithm incurs some extra flops when computing intermediate results [24, Section 5.2.2]. Consequently, the flops for `dgeqrf()` are conservative, but by a factor much less than 1.

The flop counts for the quadtree algorithm were tallied by extra code in the algorithm. Consequently, its flop counts are precise. Finding an equation to count the number of flops for the quadtree algorithm is difficult in part because it is based on some elaborate recurrence relations. Complicating the flop equation further is the Q returned by the function `e`; this Q has a particular triangular pattern to it that allows the **zero** decoration to avoid many multiplications.

5.7.2 Quadtree Algorithm Versus LAPACK Algorithm

The machines used to run these tests are described in Section 2.3.

The running times and mflop/s of `dgeqrf()` and the quadtree algorithm on the four machines are given in Figures 5.5 through 5.12.

Since it does not accumulate Q explicitly, `dgeqrf()` measures faster than the quadtree algorithm as expected. But when the plots for mflop/s are compared, the quadtree algorithm ends up outperforming `dgeqrf()`. In fact, `dgeqrf()` appears to perform quite poorly relative to the work it does on all four machines, even on the Sun machines where manufacturer's libraries were used.

On the Power Challenge (Figures 5.5 and 5.6), the quadtree QR factorization performs at a flop rate nearly as good as matrix multiplication (Figures 3.12 and 3.13)

which is easily explained by the accumulation of Q . Q is accumulated through multiplication, and comparing the flop counts in Figure 5.4, these multiplications are most of the work that the quadtree algorithm does. So it is expected that this QR factorization algorithm would perform at a flop rate nearly as good as matrix multiplication.

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

As noted in Section 5.6, `dgeqrf()` was compiled from source code and linked to the manufacturer's BLAS on the Power Challenge; the performance of `dgeqrf()` might be improved on the Power Challenge with more tiling or more information about the Power Challenge. In contrast, without any tuning the quadtree algorithm performs well, demonstrating its strength. Its performance decrease (compared to matrix multiplication) can be explained by the nature of the QR algorithm which is less memory friendly than matrix multiplication; this same explanation does not work for the magnitude of the decrease in performance of `dgeqrf()` compared to `dgemm()`.

Again the Octane (Figures 5.7 and 5.8) has interesting results. Just like the Power Challenge, `dgeqrf()` was compiled from source code and so suffers from the same decrease in performance. Also, comparing Figures 3.15 and 5.8, it is clear that striding is a an issue at order 2048 despite our compensating for it (Section 2.4.2).

Also evident in the performance of `dgeqrf()` on the Octane is its problem with the virtual memory system. Again, the page faults for `dgeqrf()` are plotted (Figure 5.7),

and they increase sharply after order 3500. The virtual-memory problem manifests itself at a larger order for `dgeqrf()` than `dgemm()` (which started having problems at order 2500) because there is only one matrix for `dgeqrf()` plus, perhaps, some scratch space; `dgemm()` needs space for at least three full matrices.

The plot of the performance of the quadtree algorithm on the Octane appears to be a step function, with each peak at an order that is a power of two. Handling the padding appears to have a greater cost on the Octane with the QR factorization algorithm. But the overall performance on the Octane is respectable. Most notably, the performance of the quadtree algorithm continues to improve in spite of the fact that another level of the hierarchy (paging) is being used. So, just as with matrix multiplication, the results on the Octane demonstrate the lesson of the quadtree matrix: *the quadtree matrix handles each level of the memory hierarchy without extra coding effort and without any knowledge of machine specifics.*

Performance on the Suns (Figures 5.9 and 5.10 for the Enterprise 450 and Figures 5.11 and 5.12) is also telling, in different ways. It is clear that there are striding problems for `dgemm()` and `dgeqrf()` at orders that are a power of two (see Figures 3.17 and 5.10 in particular). The performance decrease at these orders is not as bad for `dgeqrf()` because its overall performance on the Suns (just as it was on the SGIs) is *much* lower than it was for `dgemm()`. However, unlike the SGIs, LAPACK on the Suns is part of the Sun Performance Library. One would expect manufacturer-supplied code to be as efficient as possible, but these results do not demonstrate this. Yet the quadtree algorithm does not seem to take any performance hit at all on the Suns, even though the QR factorization is not as efficient as matrix multiplication.

Overall, the quadtree algorithm demonstrably handles all levels of the memory

hierarchy without extra or special knowledge, and the performance of the QR factorization remains close to the performance of the matrix-multiplication algorithm.

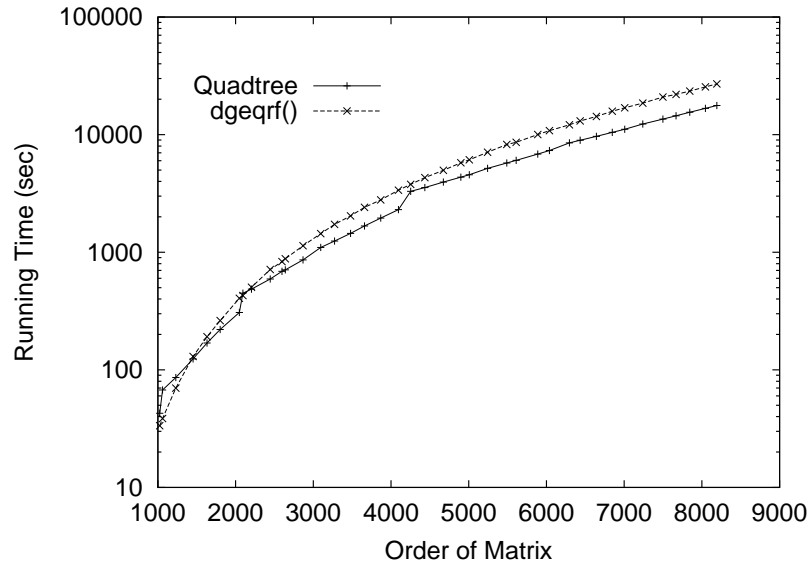


Figure 5.5: Running time of uniprocessor QR factorization on Power Challenge

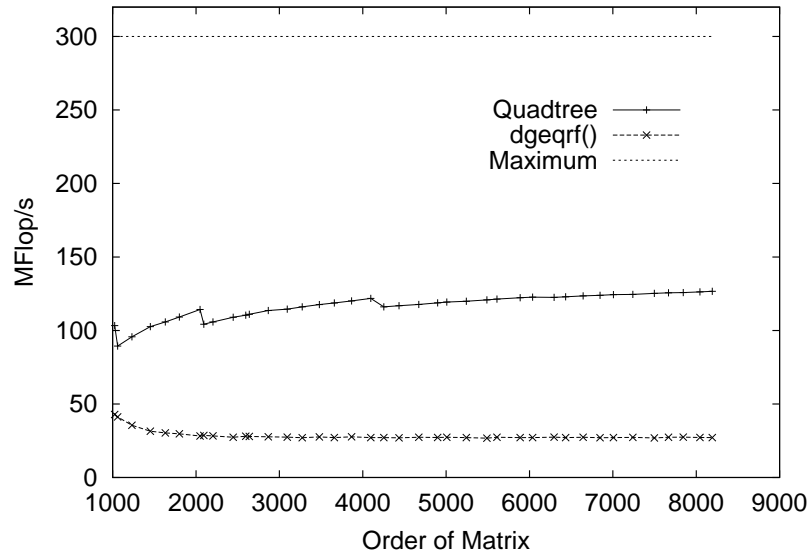


Figure 5.6: Mflop/s of uniprocessor QR factorization on Power Challenge

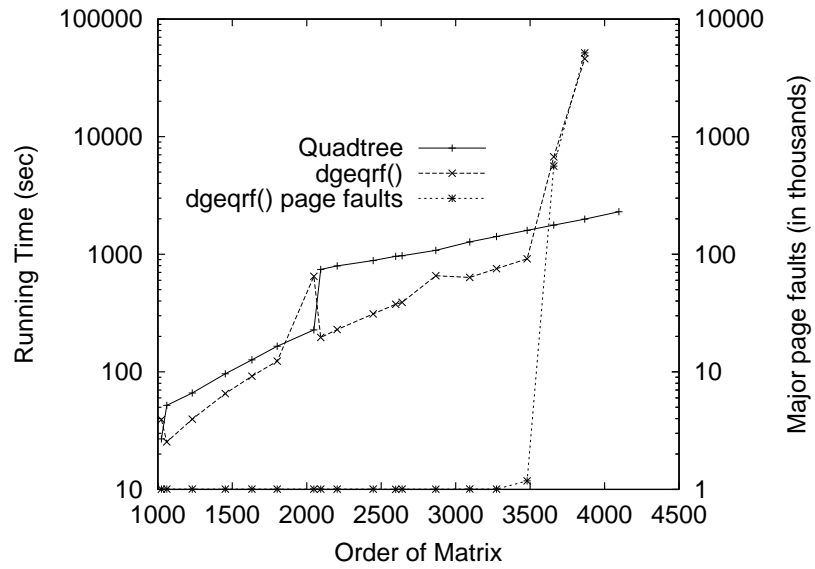


Figure 5.7: Running time of uniprocessor QR factorization on Octane

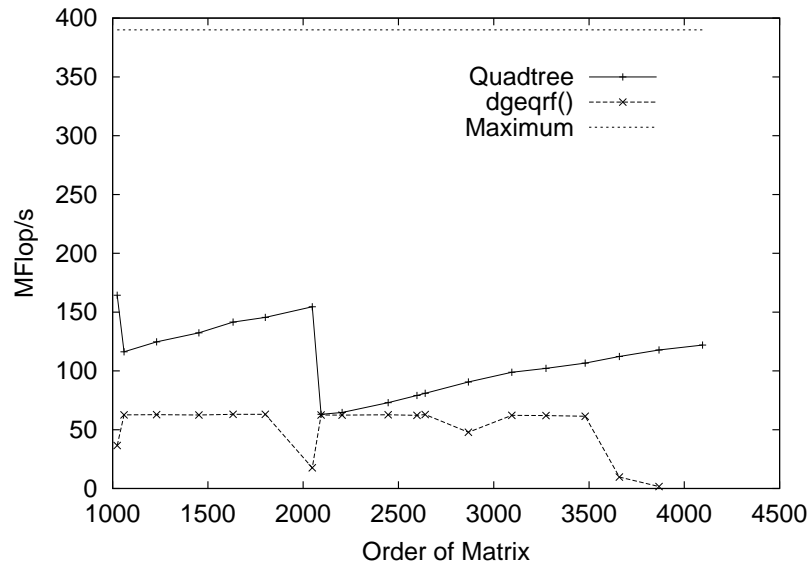


Figure 5.8: Mflop/s of uniprocessor QR factorization on Octane

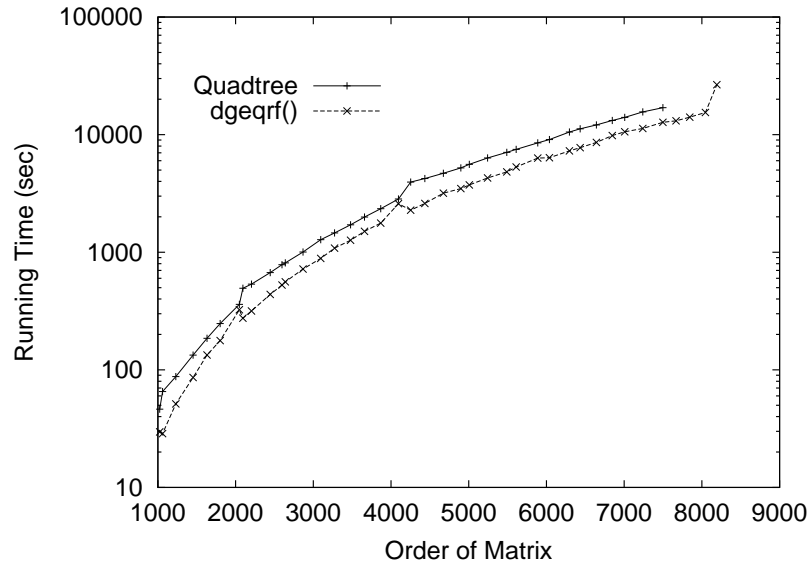


Figure 5.9: Running time of uniprocessor QR factorization on Enterprise 450

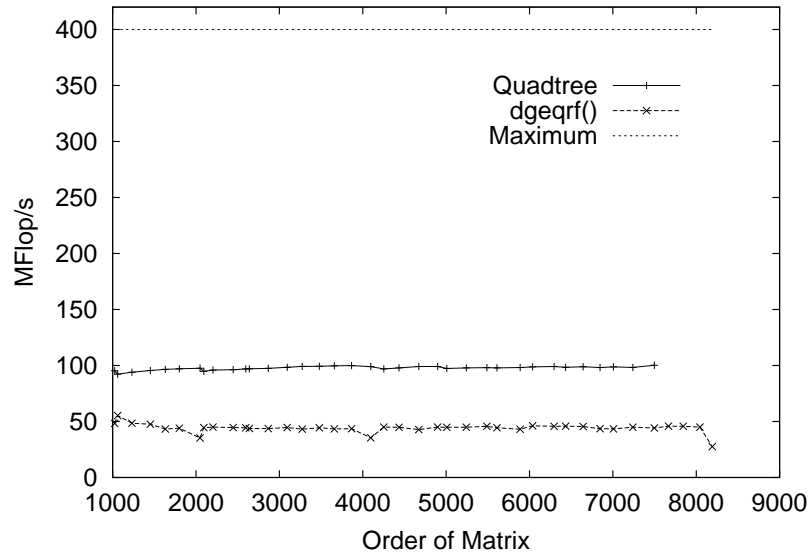


Figure 5.10: Mflop/s of uniprocessor QR factorization on Enterprise 450

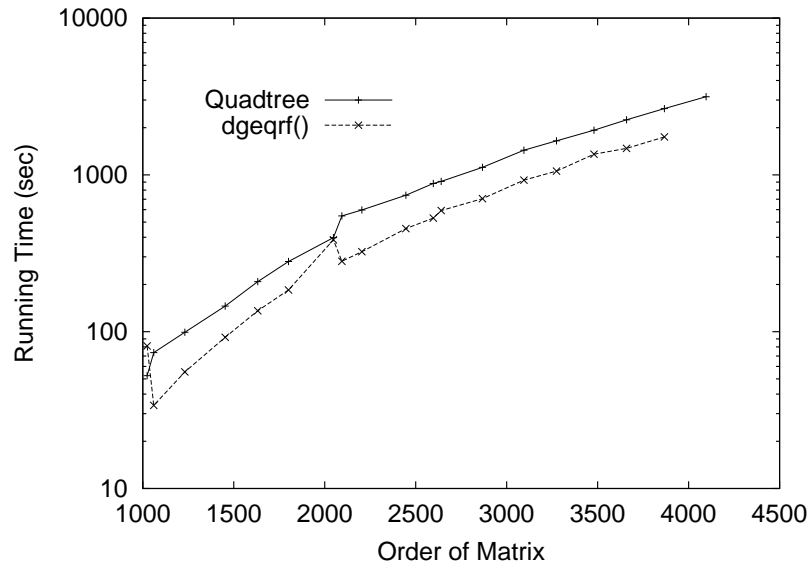


Figure 5.11: Running time of uniprocessor QR factorization on Ultra 5/10

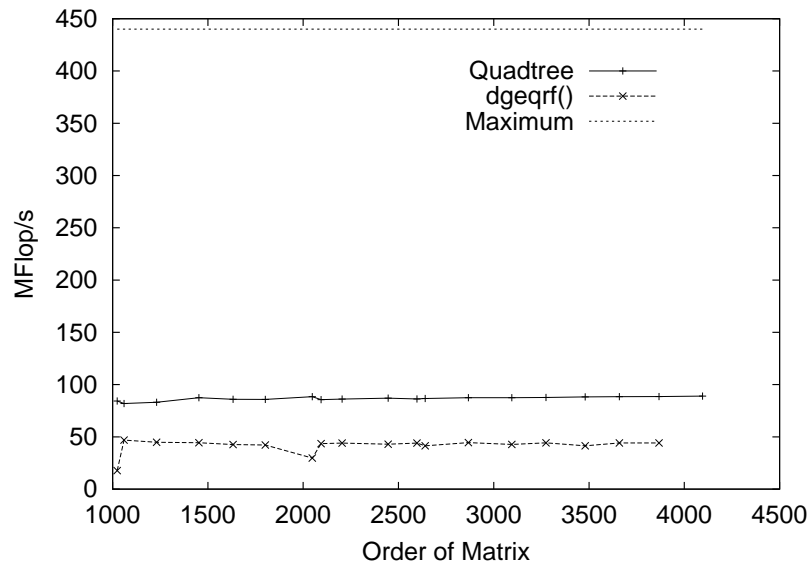


Figure 5.12: Mflop/s of uniprocessor QR factorization on Ultra 5/10

6

Parallel QR Factorization

Parallelizing the quadtree algorithm for QR factorization follows the same logic as for matrix multiplication.

6.1 QR Parallel Cases

The QR factorization algorithm is divided into cases like the matrix multiplication algorithm (Chapter 4). For QR factorization there are fewer cases to consider (even with both f and e to consider). Unfortunately, the opportunities for parallelism are fewer than with matrix multiplication. Since matrix multiplication is a fundamental part of QR factorization, however, much parallel performance of QR factorization is still available from parallel multiplication.

The parallel dispatch for each case is diagrammed in a chart similar to those in Chapter 4 (see especially Figure 4.2). These charts use the steps established in Figures 5.2 and 5.3 although the naming steps of those functions are omitted in the parallel dispatch charts.

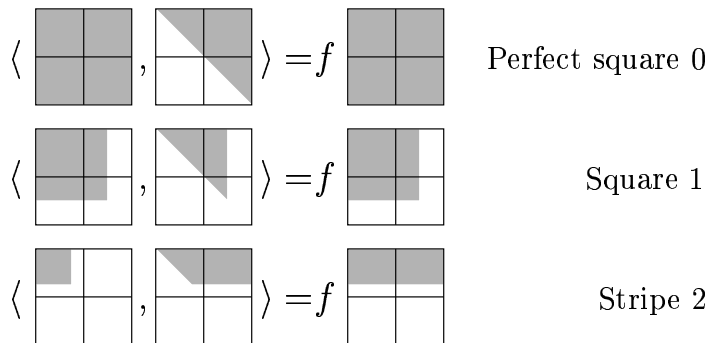


Figure 6.1: Parallel patterns of f

Often the multiplications needed to update A or Q are unbalanced. For example, Step 4 of f in the square case with minority padding consists of four multiplications, each one a different case: a perfect-square case, a stripe•square case, a column•square case, and a square case. These multiplications are not well balanced with each other and so should not be dispatched in parallel. Each one is called sequentially and given all available processors; the parallel matrix multiplication routines can still find parallelism deeper in the function-call tree. So multiplication steps in the parallel dispatch charts are annotated as “deferred” or “immediate”. “Deferred” indicates that parallel dispatch is deferred to the multiplication routines; “immediate” indicates that the multiplications of that step are dispatched in parallel immediately at the current level.

6.1.1 Parallel Patterns of f

The function f breaks down into three cases as sketched in Figure 6.1.

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 6.2.

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. This dispatch is given in Figure 6.3.

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, given in Figure 6.3.

6.1.2 Parallel Patterns of e

The function e breaks down into only two cases as depicted in Figure 6.4. The case and size of the padding is determined by the padding in S since N is always upper-triangular dense.

Step 1: $\langle Q_1, R_1 \rangle = f(A \downarrow \text{nw})$ perfect square	Step 2: $\langle Q_2, R_2 \rangle = f(A \downarrow \text{sw})$ perfect square
Step 3: $\langle Q_3, R_3 \rangle = e(R_1, R_2)$ perfect square	
Step 4: $Q_4 = Q_{1\&2} Q_3$ immediate	
Step 5: $\begin{bmatrix} U_n \\ U_s \end{bmatrix} = Q_4^T \begin{bmatrix} A \downarrow \text{ne} \\ A \downarrow \text{se} \end{bmatrix}$ immediate	
Step 6: $\langle Q_6, R_6 \rangle = f(U_s)$ perfect square	
Step 7: $Q = Q_4 \begin{bmatrix} I & Z \\ Z & Q_6 \end{bmatrix}$ immediate	

Figure 6.2: Parallel QR factorization: perfect square dispatch

Step 1: $\langle Q_1, R_1 \rangle = f(A \downarrow \text{nw})$ perfect square
Step 2: $\langle Q_2, R_2 \rangle = f(A \downarrow \text{sw})$ stripe
Step 3: $\langle Q_3, R_3 \rangle = e(R_1, R_2)$ stripe
Step 4: $Q_4 = Q_{1\&2} Q_3$ deferred
Step 5: $\begin{bmatrix} U_n \\ U_s \end{bmatrix} = Q_4^T \begin{bmatrix} A \downarrow \text{ne} \\ A \downarrow \text{se} \end{bmatrix}$ deferred
Step 6: $\langle Q_6, R_6 \rangle = f(U_s)$ square
Step 7: $Q = Q_4 \begin{bmatrix} I & Z \\ Z & Q_6 \end{bmatrix}$ deferred

Figure 6.3: Parallel QR factorization: square and stripe dispatch

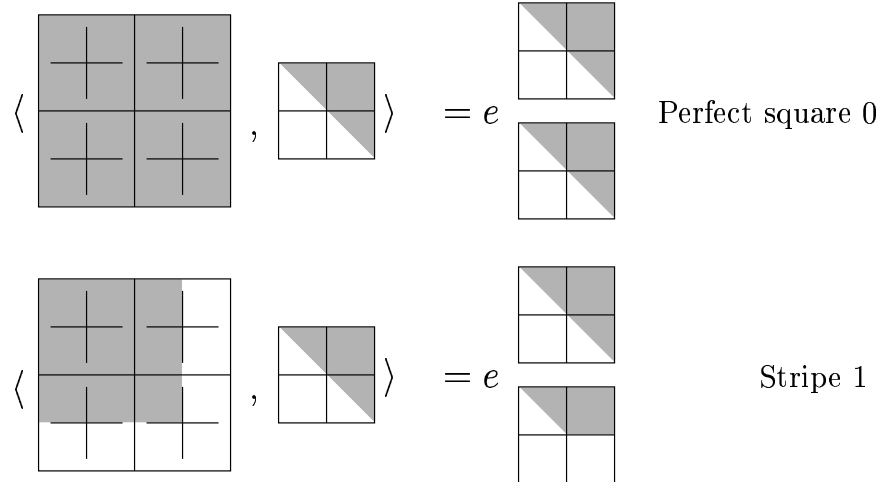


Figure 6.4: Parallel patterns of e

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 6.5.

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 (unlike the non-balanced cases of f).

MAJORITY-PADDING DISPATCH: as noted, this case does not merely reduce to an instance of itself. The steps must be done sequentially (although some are trivial). This pattern of dispatch is given in Figure 6.6.

MINORITY-PADDING DISPATCH: all steps must be done sequentially although the multiplications steps can be dispatched immediately in parallel. This pattern of dispatch is given in Figure 6.7.

Step 1: $\langle Q_1, \tilde{N}_1 \rangle = e(N \downarrow \text{nw}, S \downarrow \text{nw})$ perfect square	Step 2: $\langle Q_2, \tilde{N}_2 \rangle = e(N \downarrow \text{se}, S \downarrow \text{se})$ perfect square
Step 3: $\begin{bmatrix} U_n \\ U_s \end{bmatrix} = Q_1^T \begin{bmatrix} N \downarrow \text{ne} \\ S \downarrow \text{ne} \end{bmatrix}$ immediate	
Step 4: $\langle Q_4, R_4 \rangle = f(U_s)$ perfect square	
Step 5: $Q_5 = Q_{1\&2} \begin{bmatrix} I & Z & Z & Z \\ Z & I & Z & Z \\ Z & Z & Q_4 & Z \\ Z & Z & Z & I \end{bmatrix}$ immediate	
Step 6: $\langle Q_6, \tilde{N}_6 \rangle = e(\tilde{N}_2, R_4)$ perfect square	
Step 7: $Q = Q_5 \begin{bmatrix} I & Z & Z & Z \\ Z & Q_6 \downarrow \text{nw} & Q_6 \downarrow \text{ne} & Z \\ Z & Q_6 \downarrow \text{sw} & Q_6 \downarrow \text{se} & Z \\ Z & Z & Z & I \end{bmatrix}$ immediate	

Figure 6.5: Parallel QR elimination: perfect square dispatch

<p>Step 1: $\langle Q_1, \tilde{N}_1 \rangle = e(N \downarrow_{\text{nw}}, S \downarrow_{\text{nw}})$</p> <p>Step 2: $\langle Q_2, \tilde{N}_2 \rangle = \langle I, Z \rangle$</p> <p style="text-align: center; font-size: small;">stripe trivial</p>
<p>Step 3: $\begin{bmatrix} U_n \\ U_s \end{bmatrix} = Q_1^T \begin{bmatrix} N \downarrow_{\text{ne}} \\ S \downarrow_{\text{ne}} \end{bmatrix}$</p> <p style="text-align: center; font-size: small;">deferred</p>
<p>Step 4: $\langle Q_4, R_4 \rangle = f(U_s)$</p> <p style="text-align: center; font-size: small;">stripe</p>
<p>Step 5: $Q_5 = Q_{1\&2} \begin{bmatrix} I & Z & Z & Z \\ Z & I & Z & Z \\ Z & Z & Q_4 & Z \\ Z & Z & Z & I \end{bmatrix}$</p> <p style="text-align: center; font-size: small;">deferred</p>
<p>Step 6: $\langle Q_6, \tilde{N}_6 \rangle = e(\tilde{N}_2, R_4)$</p> <p style="text-align: center; font-size: small;">stripe</p>
<p>Step 7: $Q = Q_5 \begin{bmatrix} I & Z & Z & Z \\ Z & Q_6 \downarrow_{\text{nw}} & Q_6 \downarrow_{\text{ne}} & Z \\ Z & Q_6 \downarrow_{\text{sw}} & Q_6 \downarrow_{\text{se}} & Z \\ Z & Z & Z & I \end{bmatrix}$</p> <p style="text-align: center; font-size: small;">deferred</p>

Figure 6.6: Parallel QR elimination: stripe majority-padding dispatch

<p>Step 1: $\langle Q_1, \tilde{N}_1 \rangle = e(N \downarrow \text{nw}, S \downarrow \text{nw})$ <small style="text-align: center;">perfect square</small></p> <p>Step 2: $\langle Q_2, \tilde{N}_2 \rangle = e(N \downarrow \text{se}, S \downarrow \text{se})$ <small style="text-align: center;">stripe</small></p>
<p>Step 3: $\begin{bmatrix} U_n \\ U_s \end{bmatrix} = Q_1^T \begin{bmatrix} N \downarrow \text{ne} \\ S \downarrow \text{ne} \end{bmatrix}$ <small style="text-align: center;">immediate</small></p>
<p>Step 4: $\langle Q_4, R_4 \rangle = f(U_s)$ <small style="text-align: center;">perfect square</small></p>
<p>Step 5: $Q_5 = Q_{1\&2} \begin{bmatrix} I & Z & Z & Z \\ Z & I & Z & Z \\ Z & Z & Q_4 & Z \\ Z & Z & Z & I \end{bmatrix}$ <small style="text-align: center;">immediate</small></p>
<p>Step 6: $\langle Q_6, \tilde{N}_6 \rangle = e(\tilde{N}_2, R_4)$ <small style="text-align: center;">perfect square</small></p>
<p>Step 7: $Q = Q_5 \begin{bmatrix} I & Z & Z & Z \\ Z & Q_6 \downarrow \text{nw} & Q_6 \downarrow \text{ne} & Z \\ Z & Q_6 \downarrow \text{sw} & Q_6 \downarrow \text{se} & Z \\ Z & Z & Z & I \end{bmatrix}$ <small style="text-align: center;">immediate</small></p>

Figure 6.7: Parallel QR elimination: stripe minority-padding dispatch

6.2 Parallel LAPACK

The Sun Performance Library comes with a parallel version of QR factorization.

The Power Challenge did not have any such parallel library with QR factorization. Compiling LAPACK and linking it with the manufacturer's parallel BLAS yielded execution times that were worse than uniprocessor. The precompiled ScaLAPACK binary for the Power Challenge also yielded unsatisfactory results. Consequently, no parallel runs of `dgeqrf()` are compared on the Power Challenge.

6.3 Experimental Results

The machines used to run these tests are described in Section 2.3. Only the Power Challenge and the Sun Enterprise machines are parallel machines, so only they are reported on. Speed-ups were calculated as described in Section 2.5.1.

The Power Challenge does not come with a parallel `dgeqrf()`. Two solutions were attempted: linking LAPACK to a parallel BLAS and using a Power Challenge version of ScaLAPACK [9]. 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 6.8) are all steadily, asymptotically approaching the ideals. The speed-up is quite good for two processors and is mostly respectable for four processors. The speed-up for eight processors is disappointing, getting just over half the speed-up it should, although the increase in the speed-up is very clear. Overall, the

poorer speed-up with more processors is not unexpected, especially with the quadtree dispatch. Opportunities for parallel dispatch in the quadtree QR factorization are much rarer than they are for matrix multiplication. By the time all eight processors are used, the blocks operands are much smaller than they would be in matrix multiplication, and so the parallel payoff is much less.

On the Enterprise 450, the speed-up of `dgeqrf()` was computed using wall-clock time (see Section 2.5.2). The uniprocessor results on this machine (see Figures 5.9 and 5.10) call the the parallel speed up into question in at least two ways. First, as wall-clock times, the uniprocessor times were often quite large and so yield large parallel speed ups (as with matrix multiplication). Second and much more damaging, the uniprocessor code for `dgeqrf()` on the Enterprise 450 performs terribly as seen in its mflop/s (see Figure 5.10). It is easy to make parallel code with excellent speed ups when the uniprocessor time is terrible. So these results are highly suspect.

It is important to note that the parallel speed ups of `dgeqrf()` remains mostly constant on the Enterprise 450 (as far as they can be trusted). In contrast, the speed-up of the quadtree algorithm 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 gets better as the order increases, ending up better than the performance of `dgeqrf()`.

Overall, the parallel performance of the quadtree QR factorization algorithm is quite respectable. It is not as good as the performance of the parallel matrix multiplication algorithm, but then the opportunities for parallelism are considerably reduced with QR factorization. This performance, combined with the performance of parallel

multiplication, demonstrates that the divide-and-conquer paradigm can be successful in writing parallel algorithms.

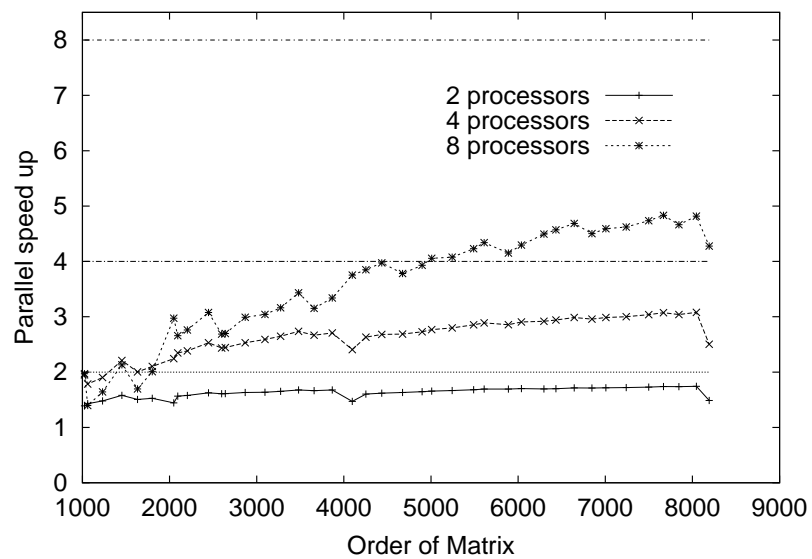


Figure 6.8: Speed-up for quadtree QR factorization on Power Challenge

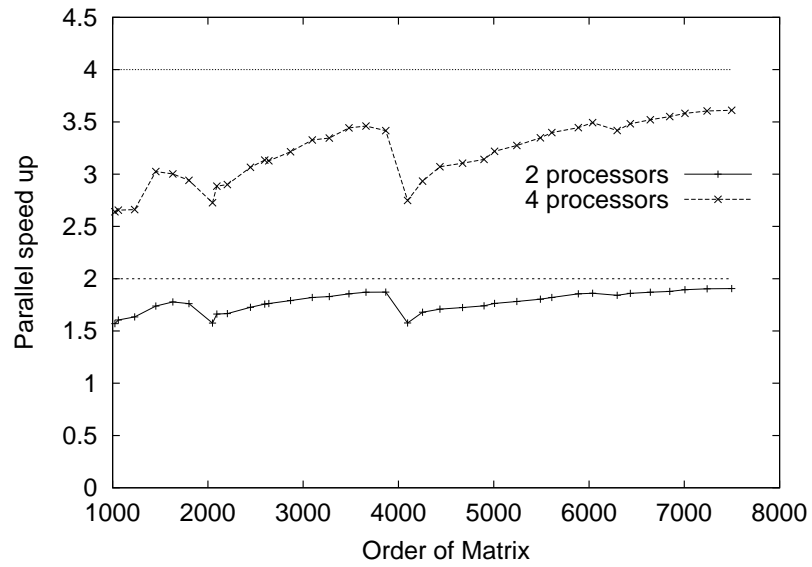


Figure 6.9: Speed-up for quadtree QR factorization on Enterprise 450

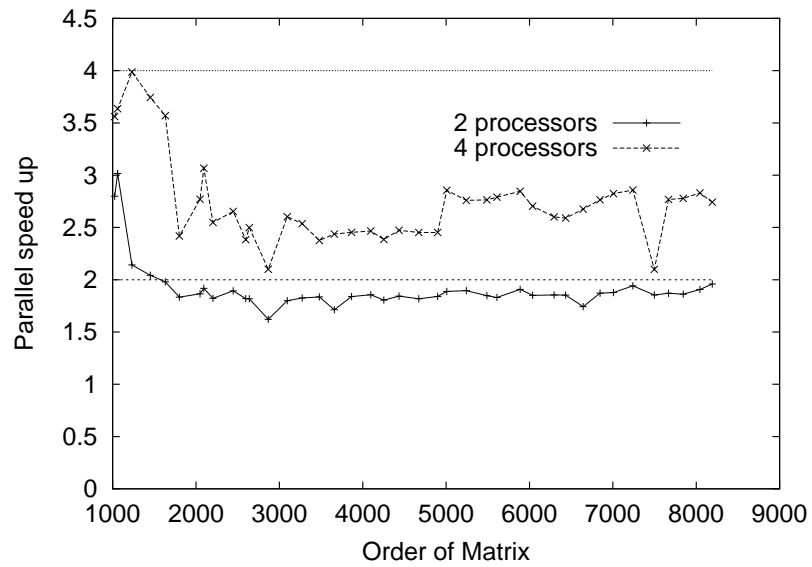


Figure 6.10: Speed-up for `dgeqrf()` on Enterprise 450

Conclusion

7.1 Results

The quadtree matrix algorithms presented in this work are not clear-cut winners over their BLAS or LAPACK equivalents. Decades of work contributing to the BLAS libraries have paid off in efficient implementations. The lessons learned from the BLAS libraries should pay off in efficient LAPACK libraries. Modern computer architectures and modern compilers have been designed with the iterative loops of those libraries in mind. More recent work suggests that the quadtree matrix multiplication can beat BLAS when more time is spent doing the compiler's job of fine tuning the algorithm [48].

It is interesting to note where BLAS and LAPACK lose. On the Octane, `dgemm()` and `dgeqrf()` lose to the quadtree matrix algorithms on large matrices (see Figures 3.14 and 5.7, respectively). Again, this does not imply that the quadtree matrix algorithms have their niche on this machine for those sizes. It means that BLAS and LAPACK were not fully tuned on that machine. Certainly they could be fixed by

tiling their algorithms for virtual memory; ATLAS [45] could be used to figure out the machine parameters. The real implication of these results is that the quadtree matrix algorithms are, in fact, tuned for the memory hierarchy. When virtual memory becomes an issue on the Octane, the quadtree matrix algorithms work for it just as well as it did for all of the other levels of the hierarchy *without changing the algorithm* and *without knowing any particulars about the Octane's memory*—cache-oblivious [22]. The quadtree algorithms *are* tuned for the memory hierarchy; and, especially in the light of more recent results [48], the performance difference is a matter of compiling recursion.

The parallel results suggest that the divide-and-conquer paradigm and quadtree matrices works rather well. While some results are disappointing (like eight processors on the Power Challenge, Figure 6.8), *all* of the speed-ups increased as the order of the matrix increased.

7.2 Comments

While working on this research, I became more and more bothered by what I saw as the typical solution for the memory hierarchy and row-major matrices. Typically a solution (like tiling) is presented for the cache. The solution involves changing the algorithm in significant ways, always with knowledge about the size of the cache. Papers with these solutions then give a hand-wave suggestion that the rest of the memory hierarchy can be dealt with similarly. But that hand-wave hides too much work. The algorithm must be changed for every level of the hierarchy with knowledge about each level. Now, granted some of this process can be automated by a compiler,

but then at the very least this requires a compilation (perhaps lengthy depending on what parameters need to be tuned) every time the machine's memory changes in some way.

Solutions for the quadtree matrix algorithms are never conjured in terms of the cache; they are solutions for the *entire* hierarchy all at once because the matrices are thought of in terms of quadrants. Since the transfer blocks in *each* level of the memory hierarchy is a quadrant, thinking in terms of quadrants is all that is needed to work on every different memory hierarchy (cf. Section 1.7). This is demonstrated most clearly on the Octane (Figures 3.14, 3.15, 5.7, and 5.8) where two levels of the hierarchy come into play. The quadrant algorithm handles them with ease without consciously being designed for that particular machine with its particular configuration.

I was even more disappointed (but not surprised) by the lack of optimization for recursive functions in the C compiler. Solutions are known [42, 10]. I had to write some rather ugly code to help the optimizer.

7.3 Future Work

One of the most exciting things about this research is the wealth of possible follow-up projects.

7.3.1 Rectangular Matrices

The algorithms presented in this work should be expanded to handle rectangular matrices. This should not affect the uniprocessor code or its performance; but

the parallel dispatch would be greatly affected since much of the balancing is done assuming that the colonnade in one quadrant is equivalent to the stripe in another. One thing to consider in this dispatch is how close the dimensions are (similar to the padding size of Section 4.3.1); for example, a 7×16 stripe is roughly equivalent to a 16×8 colonnade (especially with an 8×8 base case).

7.3.2 Processor Dispatch

Another consideration is distributing processors proportionally, not just simply in half. For example, each parallel multiplication could be split in four, one for each quadrant of C . Workloads for each process could be computed based on the shapes and sizes of the quadrants. Then the processors are handed out to the processes based on these workloads. It is hard to honor the workload ratios, though, when the number of processors is small. For example, consider the quicksort algorithm from Chapter 1 (Figure 1.3). If there were three processors available but `upper` and `lower` had about the same length, there is no clean way to split the three processors over the two recursive calls to `qsort` as written in Figure 1.3. One possibility would be to have two pivots that partition the original list into three sublists; this solution requires rewriting the algorithm, and only handles a limited number of cases. It can be generalized, but not trivially and, I would argue, not simply. Other solutions may also exist. Perhaps a simple solution is not possible. Regardless, the open question is how these other solutions would translate into solutions for quadtree matrix algorithms.

Computing the workload metric is also hard since it must consider (at least) the amount of computation involved as well as the amount and shape of the quadrants (in terms of memory). The quicksort algorithm is misleading here since the workload

just comes down to the length of the lists. But with quadtree matrices, the area of the matrices being worked on must also be considered. For example, the colonnade•stripe and stripe•colonnade cases in matrix multiplication (see Figure 4.1) are computationally equivalent, but the amount of memory they access (due to C) is quite different even if A and B have equivalent dimensions. Consequently, the memory accesses will be different, leading to different execution times. All of this must be considered in a workload metric.

7.3.3 Concise Representation of Q

A more concise representation of Q in QR factorization may also be useful. The LAPACK routines store Q in the lower triangular portion of R . The quadtree matrix algorithm could do the same, however I have concerns about how the updates to A would be handled, perhaps losing its blocked nature.

7.3.4 Sparse Matrices

The linked quadtree matrix has been touted as a good sparse matrix representation; however, the linked quadtree matrix saves space best when large quadrants are zero. This is one reason why the zero padding is put as large colonnade and stripe blocks on the east and south, respectively; the space savings for that padding is logarithmic in the size of the padding. But a single element in a quadrant requires *some* tree structure to store it. The savings is not great in sparse matrices where elements are scattered about the matrix.

While playing around with the mathematics of Morton-order indices (see Wise [47]),

I wondered about using them as indices in a sparse matrix representation similar to the standard Compressed Row Storage (CRS) and Compressed Column Storage (CCS) [5, 15]. Both CRS and CCS use explicit Cartesian coordinates to store non-zero elements of a sparse matrix. CRS stores elements row by row, CCS column by column. I believe Morton-order indices could be used similarly, storing elements by quadrant. While this CMS (Compressed Morton Storage) would be as memory efficient as CRS or CCS, it may not be as computationally efficient or as easy to use.

7.3.5 Other Algorithms

One of the most important future projects is working out other quadtree matrix algorithms. This would include Strassen's formulation of matrix multiplication that reduces the number of multiplications at the cost of more additions and increased error. Other algorithms include many direct factorizations like LU factorization and Cholesky factorization as well as indirect algorithms like the Jacobi and Gauss-Seidel iterations. There are also algorithms (both direct and indirect) for finding eigenvalues, for solving the least squares problem, and for a variety of other problems. A good first step would be to work on a BLAS library for quadtree matrices and binary vectors, then move on to an LAPACK library. Then work through Golub and Van Loan [24].

Iterative methods [24, Chapter 10], which iteratively apply matrix-vector multiplication to A to solve $Ax = b$ for x , could be easily implemented with quadtree matrices and some simple kernel operations (like matrix-vector multiplication). Usually iterative methods are used on sparse matrices.

7.3.6 Patterns in Algorithms

Exploring rectangular matrices and other algorithms is quite important for recognizing patterns in the algorithms:

- The up/down pattern of matrix multiplication may generalize well for algorithms with a lot of quadrant sharing.
- QR factorization uses two functions. How many other problems need mutually recursive functions in this way?
- Multiplication parallelizes quite well and has many cases; the two functions of QR factorization have few cases and little parallelism. Correlation or coincidence?

Rectangular matrices would undoubtedly complicate the parallel dispatch perhaps revealing other patterns.

7.4 Curious Observations

One observation from the QR factorization functions has really intrigued me: Step 2 of function f (see Figure 5.2) is actually unnecessary if function e were rewritten to take an upper triangular N while S is still an unfactored stripe (or perfect square). However, this would eliminate nearly all of the independent divide-and-conquer in these functions. Steps 1 and 2 are the only steps of f that can be done in parallel against each other; drop Step 2, and f defers all of its parallelism. If e were changed to take a stripe S , it would be called recursively four times (once for each quadrant

$e_m(N \downarrow \mathbf{nw}, S \downarrow \mathbf{sw})$	
$e_m(N \downarrow \mathbf{nw}, S \downarrow \mathbf{nw})$	$e_m(N \downarrow \mathbf{se}, S \downarrow \mathbf{se})$
$e_m(N \downarrow \mathbf{se}, S \downarrow \mathbf{ne})$	

Figure 7.1: Modified parallel QR elimination (perfect square) dispatch

of S) where only *two* of those calls could be done in parallel; the dispatch for this modified e (i.e., e_m) is presented in Figure 7.1.

It also strikes me that the parallelism is quite obvious in the versions of Section 5.2. The parallelism in Figure 7.1 is not immediately obvious.

The most intriguing question that this observation raises is the principle at work here: what coding aphorism should we give to our programmers? “Do whatever work you can, even if it gets you only partway to a solution”? “Write symmetric algorithms”?

The QR factorization has a few interesting cases to explore:

- The square and stripe cases of f with minority padding result in the same parallel dispatches.
- Why does e not have a parallel square case although f and multiplication do? (Incidentally, e would have a square case if A were rectangular and tall.)

7.5 Conclusion

As laid out in Section 1.1, before the divide-and-conquer paradigm can be taken seriously as a solution for high-performance computing, two problems must be solved:

parallelism and efficient memory reuse.

For this dissertation, I have implemented algorithms for matrix multiplication and QR factorization for quadtree matrices. The experimental results (Sections 3.5.3, 4.4, 5.7.2, and 6.3) are very promising, even inspiring. The quadtree-matrix algorithm for matrix multiplication is competitive with its BLAS equivalent, especially the parallel speed-ups. The quadtree-matrix algorithm for QR factorization even outperforms its LAPACK equivalent on one processor and on multiple processors.

Efficient memory use of the quadtree matrix is clearly demonstrated by the results on the SGI Octane (see Figures 3.14 and 3.15 for matrix multiplication, Figures 5.7 and 5.8 for QR factorization). The BLAS and LAPACK functions fail miserably once virtual memory starts being used; those libraries were not tuned on that machine. The quadtree algorithms, though, handle the virtual memory without any problems *without changing the code or knowing any machine-specific parameters*—justification for the claims made in Section 1.7. Adapting BLAS and LAPACK for the Octane's virtual memory would require changing the code and adding machine-specific information.

Effective parallelism of the divide-and-conquer programming paradigm is demonstrated on both of the parallel machines in Sections 4.4 and 6.3. The quadtree-matrix algorithms were usually as good, if not better, than the BLAS and LAPACK equivalents in terms of their parallel speedups. The quadtree algorithm showed either obvious asymptotic growth to the ideal speed-ups, or its performance was right at the ideal.

The BLAS and LAPACK libraries have had the benefit of decades of research; these decades have paid off in optimizing compilers tuned for the needs of these

libraries and in computer architectures built with their needs in mind. Quadtree matrices for high-performance computing are comparatively much younger, and so do not enjoy the same benefits. If quadtree matrices had the same amount of research, optimizing compilers would do a better job with recursion and computer architectures could do arithmetic with Morton-order indices [47] as basic integer operations.

This work demonstrates that even as young as they are, quadtree matrices show a qualified success for both parallelism and memory use. Even without quadtree-aware optimizing compilers, the quadtree-matrix algorithms are competitive and even beat the older libraries. More importantly, the quadtree-matrix algorithm does it without any knowledge of the target machine. In the larger context, these results also indicate usefulness of the divide-and-conquer paradigm for high-performance computing.

A

Proofs

Section 5.4 makes several claims about the correctness of the QR functions in Section 5.2. This appendix proves those claims.

There are three postconditions for both f and e that need to be proved: Q is orthogonal, R is upper-triangular, and they compute what they claim to compute. Furthermore, it is also necessary to show that the transformations are safe and that a nonsingular matrix A will be successfully factored.

The first two sections of this appendix first prove some useful lemmas about transposes and orthogonal matrices.

A.1 Transpose Properties

This first lemma establishes some simple transpose properties:

Lemma A.1 *Let A and B be $n \times n$ matrices. Then the following properties hold:*

$$(AB)^T = B^T A^T$$

$$(A^T)^T = A$$

$$(A^{-1})^T = (A^T)^{-1}$$

Proof The first equation is proved by Friedberg et al. [21, p. 74]. The second equation is somewhat trivial and is given without proof. The third equation is stated by Golub and Van Loan [24, Section 2.1.3]. \square

A.2 Lemmas for Orthogonal Matrices

It helps to have a few lemmas to prove that the Q s resulting from the QR factorization functions f and e from Chapter 5 are orthogonal.

Lemma A.2 *Let Q_1, Q_2, \dots, Q_m be orthogonal $n \times n$ matrices. Then the product $Q = \prod_{i=1}^m Q_i$ is orthogonal.*

Proof by induction.

Base case. Let $m = 0$. The product is then the identity matrix, the multiplicative identity, which is clearly orthogonal.

Induction case. Let Q_1, Q_2, \dots, Q_{m+1} be orthogonal $n \times n$ matrices. Let $Q = \prod_{i=1}^{m+1} Q_i$. Assume that the lemma holds for the product of m orthogonal

matrices. Consequently, the product $\hat{Q} = \prod_{i=1}^m Q_i$ is orthogonal. Then Q can be written as the product of two orthogonal matrices:

$$Q = \prod_{i=1}^{m+1} Q_i = \left(\prod_{i=1}^m Q_i \right) Q_{m+1} = \hat{Q} Q_{m+1}.$$

Since \hat{Q} is orthogonal by the induction hypothesis and Q_{m+1} is given as orthogonal, Q is orthogonal:

$$Q^T Q = (\hat{Q} Q_{m+1})^T \hat{Q} Q_{m+1} = Q_{m+1}^T \hat{Q}^T \hat{Q} Q_{m+1} = Q_{m+1}^T I Q_{m+1} = I. \quad \square$$

A block-diagonal matrix with orthogonal blocks along the main diagonal is orthogonal.

Lemma A.3 *Let Q_m and Q_n be $m \times m$ and $n \times n$ orthogonal matrices, respectively. Then the matrix*

$$\begin{bmatrix} Q_m & Z \\ Z & Q_n \end{bmatrix}$$

is orthogonal.

Proof Let Q_m and Q_n be $m \times m$ and $n \times n$ orthogonal matrices, respectively. Then

$$\begin{bmatrix} Q_m^T & Z \\ Z & Q_n^T \end{bmatrix} \begin{bmatrix} Q_m & Z \\ Z & Q_n \end{bmatrix} = \begin{bmatrix} Q_m^T Q_m & Z \\ Z & Q_n^T Q_n \end{bmatrix} = \begin{bmatrix} I & Z \\ Z & I \end{bmatrix} = I. \quad \square$$

Lemma A.3 is general enough to cover any case, including many that do not make sense as quadtree matrices. The lemma can be applied multiple times for block-diagonal matrices with more than two blocks on the diagonal.

A third lemma handles permutation matrices. Paraphrased just slightly from Golub and Van Loan [24, Section 3.4.1]:

Definition A.1 *A permutation matrix is just the identity matrix with its stripes permuted.*

Lemma A.4 *A permutation matrix is orthogonal.*

Proof See Section 3.4.1 of Golub and Van Loan [24]. □

The definition of an inverse in Definition 1.6 actually just defines what is known as a *right inverse*. Since matrices do not commute in general, the following lemma is non-trivial:

Lemma A.5 *If Q is orthogonal, then $QQ^T = I$.*

Proof By the definition of an orthogonal matrix (Definition 1.7), if Q is orthogonal, then $Q^T Q = I$. So, by the definition of a (right) inverse (Definition 1.6), the inverse of Q^T is Q ; symbolically, $(Q^T)^{-1} = Q$. Taking the transpose of both sides, the equation becomes $Q^T = ((Q^T)^{-1})^T$. Using the properties in Lemma A.1, this equation can be transformed as follows:

$$Q^T = ((Q^T)^{-1})^T = ((Q^T)^T)^{-1} = Q^{-1}.$$

So Q^T is the inverse of Q , thus $QQ^T = I$ by the definition of a matrix inverse.

□

A.3 Correctness of QR Factorization

With the lemmas from the previous section, the functions f and e for QR factorization (Chapter 5) can be proven in one theorem, copied from Section 5.4.1:

Theorem 5.1 *This theorem consists of two clauses:*

Clause F *If $f: A_f \mapsto \langle Q_f, R_f \rangle$ is defined as in Figure 5.2 and A_f is an $n \times n$ matrix, then Q_f is an $n \times n$ orthogonal matrix, R_f is an $n \times n$ upper-triangular matrix, and $A_f = Q_f R_f$.*

Clause E *If $e: \langle N_e, S_e \rangle \mapsto \langle Q_e, \tilde{N}_e \rangle$ is defined as in Figure 5.3 and both N_e and S_e are $n \times n$ upper-triangular matrices, then Q_e is a $2n \times 2n$ orthogonal matrix, \tilde{N}_e is an $n \times n$ upper-triangular matrix, and*

$$\begin{bmatrix} N_e \\ S_e \end{bmatrix} = Q_e \begin{bmatrix} \tilde{N}_e \\ Z \end{bmatrix}.$$

Proof by mutual induction.

Base case of Clause F. Suppose $A_f = [a_{11}]$ is a 1×1 matrix. Then $Q_f = I = [1]$, and $R_f = A_f = [a_{11}]$. Q_f is orthogonal; R_f is upper-triangular; both Q_f and R_f are 1×1 matrices; and $Q_f R_f = I A_f = A_f$.

Base case of Clause E. There are two cases:

First, if S_e is an $n \times n$ zero matrix, then Q_e is the $2n \times 2n$ identity matrix and

$\tilde{N}_e = N_e$. Q_e is clearly orthogonal; \tilde{N}_e is upper-triangular and $n \times n$; and

$$Q \begin{bmatrix} \tilde{N}_e \\ Z \end{bmatrix} = I \begin{bmatrix} \tilde{N}_e \\ Z \end{bmatrix} = \begin{bmatrix} N_e \\ S_e \end{bmatrix}.$$

The second base case for Clause E is when S_e is not a zero matrix and $n = 1$. For the ease of notation, let $N_e = [a]$ and $S_e = [b]$. Equation 5.1 defines Q_e as follows:

$$Q_e = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}, \quad \text{where } c = \frac{a}{\sqrt{a^2 + b^2}} \quad \text{and} \quad s = \frac{-b}{\sqrt{a^2 + b^2}}.$$

\tilde{N}_e is then computed by

$$\begin{bmatrix} \tilde{N}_e \\ Z \end{bmatrix} = Q_e^T \begin{bmatrix} N_e \\ S_e \end{bmatrix} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} ca - sb \\ sa + cb \end{bmatrix}.$$

In short, $\tilde{N}_e = [ca - sb]$, which is upper-triangular and 1×1 .

Q_e is orthogonal:

$$Q_e^T Q_e = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} c^2 + s^2 & cs - cs \\ cs - cs & c^2 + s^2 \end{bmatrix} = \begin{bmatrix} c^2 + s^2 & 0 \\ 0 & c^2 + s^2 \end{bmatrix}.$$

Substituting values for c and s ,

$$c^2 + s^2 = \frac{a^2}{a^2 + b^2} + \frac{b^2}{a^2 + b^2} = \frac{a^2 + b^2}{a^2 + b^2} = 1.$$

So $Q_e^T Q_e = I$, and thus Q_e is orthogonal. Q_e is also 2×2 .

As for the computation of the base case of e ,

$$\begin{aligned}
Q_e \begin{bmatrix} \tilde{N}_e \\ Z \end{bmatrix} &= \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} ca - sb \\ 0 \end{bmatrix} = \begin{bmatrix} c^2 a - csb \\ s^2 b - csa \end{bmatrix} \\
&= \begin{bmatrix} a \left(\frac{a}{\sqrt{a^2+b^2}} \right)^2 - b \left(\frac{a}{\sqrt{a^2+b^2}} \right) \left(\frac{-b}{\sqrt{a^2+b^2}} \right) \\ b \left(\frac{-b}{\sqrt{a^2+b^2}} \right)^2 - a \left(\frac{a}{\sqrt{a^2+b^2}} \right) \left(\frac{-b}{\sqrt{a^2+b^2}} \right) \end{bmatrix} \\
&= \begin{bmatrix} a \left(\frac{a^2+b^2}{a^2+b^2} \right) \\ b \left(\frac{a^2+b^2}{a^2+b^2} \right) \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} N_e \\ S_e \end{bmatrix}.
\end{aligned}$$

Hence, the base case for e is computationally correct.

Induction case of Clause F. Consider $f: A_f \mapsto \langle Q_f, R_f \rangle$ for $n \times n$ matrix A_f . Assume that the theorem holds for f and e for all $m \times m$ matrices A , N , and S where $m < n$. All of the numbered matrices in this part of the proof are taken from the definition of f in Figure 5.2. This part of the proof then follows the steps of Figure 5.2:

Step 1. $A_f \downarrow \mathbf{nw}$ is an $n/2 \times n/2$ matrix, and $\langle Q_1, R_1 \rangle = f(A_f \downarrow \mathbf{nw})$. The induction hypothesis says that Q_1 is an $n/2 \times n/2$ orthogonal matrix, R_1 is an $n/2 \times n/2$ upper-triangular matrix, and $A_f \downarrow \mathbf{nw} = Q_1 R_1$.

Step 2. Similar to the previous step, $A_f \downarrow \mathbf{sw}$ is an $n/2 \times n/2$ matrix, and $\langle Q_2, R_2 \rangle = f(A_f \downarrow \mathbf{sw})$. The induction hypothesis says that Q_2 is an $n/2 \times n/2$ orthogonal matrix, R_2 is an $n/2 \times n/2$ upper-triangular matrix, and $A_f \downarrow \mathbf{sw} = Q_2 R_2$.

Naming of $Q_{1\&2}$. The matrix $Q_{1\&2}$ is a block-diagonal matrix:

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

Steps 1 and 2 show that Q_1 and Q_2 are $n/2 \times n/2$ orthogonal matrices, so, by Lemma A.3, this matrix is orthogonal. It is also $n \times n$.

Step 3. Since R_1 and R_2 are $n/2 \times n/2$ upper-triangular matrices from Steps 1 and 2, respectively, and since $\langle Q_3, R_3 \rangle = e(R_1, R_2)$, the induction hypothesis says that Q_3 is an $n \times n$ orthogonal matrix, R_3 is an $n/2 \times n/2$ upper-triangular matrix, and

$$\begin{bmatrix} R_1 \\ R_2 \end{bmatrix} = Q_3 \begin{bmatrix} R_3 \\ Z \end{bmatrix}.$$

Step 4. $Q_{1\&2}$ is orthogonal as shown above; Q_3 is orthogonal from the previous step. So $Q_4 = Q_{1\&2}Q_3$ is orthogonal by Lemma A.2. Since $Q_{1\&2}$ is $n \times n$ from its naming step and Q_3 is $n \times n$ from Step 3, Q_4 is also $n \times n$.

Step 5. This step updates the eastern colonnade of A_f , and so computes $n/2 \times n/2$ matrices U_n and U_s :

$$\begin{bmatrix} U_n \\ U_s \end{bmatrix} = Q_4^T \begin{bmatrix} A_f \downarrow \mathbf{ne} \\ A_f \downarrow \mathbf{se} \end{bmatrix}.$$

Multiplying both sides by Q_4 ,

$$Q_4 \begin{bmatrix} U_n \\ U_s \end{bmatrix} = Q_4 Q_4^T \begin{bmatrix} A_f \downarrow \mathbf{ne} \\ A_f \downarrow \mathbf{se} \end{bmatrix} = \begin{bmatrix} A_f \downarrow \mathbf{ne} \\ A_f \downarrow \mathbf{se} \end{bmatrix}.$$

The last equality is valid by Lemma A.5.

Step 6. Since U_s is an $n/2 \times n/2$ matrix and since $\langle Q_6, R_6 \rangle = f(U_s)$, the induction hypothesis says that Q_6 is an $n/2 \times n/2$ orthogonal matrix, R_6 is an $n/2 \times n/2$ upper-triangular matrix, and $U_s = Q_6 R_6$.

Step 7. This step defines Q_f :

$$Q_f = Q_4 \begin{bmatrix} I & Z \\ Z & Q_6 \end{bmatrix}.$$

From Step 6, Q_6 is orthogonal; so by Lemma A.3, the padded matrix involving Q_6 is also orthogonal. Q_4 is orthogonal from Step 4. So Q_f , as the product of orthogonal matrices, is itself orthogonal by Lemma A.2. Q_4 is $n \times n$ from Step 4; Q_6 is $n/2 \times n/2$ from Step 6, but its factor is padded to $n \times n$; thus Q_f is $n \times n$.

Naming of R_f . R_f is defined to be

$$R_f = \begin{bmatrix} R_3 & U_s \\ Z & R_6 \end{bmatrix}.$$

From Step 3, R_3 is upper-triangular; from Step 6, R_6 is upper-triangular. Since $R_f \downarrow \mathbf{sw} = Z$, the matrix R_f is upper-triangular. R_3 is $n/2 \times n/2$

from Step 3; U_s is $n/2 \times n/2$ from Step 5; R_6 is $n/2 \times n/2$ from Step 6; thus R_f is $n \times n$.

Step 7 establishes that Q_f is an $n \times n$ orthogonal matrix. The naming of R_f establishes that R_f is an $n \times n$ upper-triangular matrix. It only remains to show that $A_f = Q_f R_f$.

$$\begin{aligned} Q_f R_f &= Q_4 \begin{bmatrix} I & Z \\ Z & Q_6^T \end{bmatrix} \begin{bmatrix} R_3 & U_n \\ Z & R_6 \end{bmatrix} \\ &= Q_4 \begin{bmatrix} R_3 & U_n \\ Z & Q_6 R_6 \end{bmatrix} = Q_4 \begin{bmatrix} R_3 & U_n \\ Z & U_s \end{bmatrix} \quad \text{by Step 6.} \end{aligned}$$

It is now easiest to split the rightmost matrix in two by colonnades. The west colonnade takes just a bit of work:

$$\begin{aligned} Q_4 \begin{bmatrix} R_3 \\ Z \end{bmatrix} &= \begin{bmatrix} Q_1 & Z \\ Z & Q_2 \end{bmatrix} Q_3 \begin{bmatrix} R_3 \\ Z \end{bmatrix} \quad \text{by Step 4} \\ &= \begin{bmatrix} Q_1 & Z \\ Z & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ R_2 \end{bmatrix} \quad \text{by Step 3} \\ &= \begin{bmatrix} Q_1 R_1 \\ Q_2 R_2 \end{bmatrix} = \begin{bmatrix} A_f \downarrow \text{nw} \\ A_f \downarrow \text{sw} \end{bmatrix} \quad \text{by Steps 1 and 2.} \end{aligned}$$

The east colonnade is easy:

$$Q_4 \begin{bmatrix} U_n \\ U_s \end{bmatrix} = \begin{bmatrix} A_f \downarrow \mathbf{ne} \\ A_f \downarrow \mathbf{se} \end{bmatrix} \quad \text{by Step 5.}$$

Combining these two colonnade derivations, the result is $Q_f R_f = A_f$, establishing the computational correctness of the induction case of f .

So, the induction case of f is proven for all of its postconditions.

Induction case of Clause E. Consider $e: \langle N_e, S_e \rangle \mapsto \langle Q_e, \tilde{N}_e \rangle$ for $n \times n$ matrices N_e and S_e . Assume that the theorem holds for f and e for all $m \times m$ matrices A , N , and S where $m < n$. All of the numbered matrices in this part of the proof are taken from the definition of e in Figure 5.3, and this part of the proof follows the steps in that definition:

Step 1. By the definition of an upper-triangular matrix, $N_e \downarrow \mathbf{nw}$ and $S_e \downarrow \mathbf{nw}$ are $n/2 \times n/2$ upper-triangular matrices. This step computes $\langle Q_1, \tilde{N}_1 \rangle = e(N_e \downarrow \mathbf{nw}, S_e \downarrow \mathbf{nw})$. So the induction hypothesis says that Q_1 is an $n \times n$ orthogonal matrix, \tilde{N}_1 is an $n/2 \times n/2$ upper-triangular matrix, and

$$\begin{bmatrix} N_e \downarrow \mathbf{nw} \\ S_e \downarrow \mathbf{nw} \end{bmatrix} = Q_1 \begin{bmatrix} \tilde{N}_1 \\ Z \end{bmatrix}.$$

Step 2. Similar to the previous step, by the definition of an upper-triangular matrix, $N_e \downarrow \mathbf{se}$ and $S_e \downarrow \mathbf{se}$ are $n/2 \times n/2$ upper-triangular matrices. This step computes $\langle Q_2, \tilde{N}_2 \rangle = e(N_e \downarrow \mathbf{se}, S_e \downarrow \mathbf{se})$. So the induction hypothesis

says that Q_2 is an $n \times n$ orthogonal matrix, \tilde{N}_2 is an $n/2 \times n/2$ upper-triangular matrix, and

$$\begin{bmatrix} N_e \downarrow \text{se} \\ S_e \downarrow \text{se} \end{bmatrix} = Q_2 \begin{bmatrix} \tilde{N}_2 \\ Z \end{bmatrix}.$$

Naming $Q_{1\&2}$. From Steps 1 and 2, Q_1 and Q_2 are both orthogonal. Define a block-diagonal matrix M :

$$M = \begin{bmatrix} Q_1 \downarrow \text{nw} & Q_1 \downarrow \text{ne} & Z & Z \\ Q_1 \downarrow \text{sw} & Q_1 \downarrow \text{se} & Z & Z \\ Z & Z & Q_2 \downarrow \text{nw} & Q_2 \downarrow \text{ne} \\ Z & Z & Q_2 \downarrow \text{sw} & Q_2 \downarrow \text{se} \end{bmatrix}.$$

M is orthogonal by Lemma A.3. Select a permutation matrix:

$$P = \begin{bmatrix} I & Z & Z & Z \\ Z & Z & I & Z \\ Z & I & Z & Z \\ Z & Z & Z & I \end{bmatrix}.$$

Verbally, the permutation matrix P , when multiplied on the left, exchanges the second and third stripes; when multiplied on the right, it exchanges the second and third colonnades. By Lemma A.4, P is orthogonal. Thus,

$Q_{1\&2}$ can be expressed as the product of orthogonal matrices:

$$Q_{1\&2} = PMP = \begin{bmatrix} Q_1 \downarrow \mathbf{nw} & Z & Q_1 \downarrow \mathbf{ne} & Z \\ Z & Q_2 \downarrow \mathbf{nw} & Z & Q_2 \downarrow \mathbf{ne} \\ Q_1 \downarrow \mathbf{sw} & Z & Q_1 \downarrow \mathbf{se} & Z \\ Z & Q_2 \downarrow \mathbf{sw} & Z & Q_2 \downarrow \mathbf{se} \end{bmatrix}.$$

Lemma A.2 indicates that $Q_{1\&2}$ is orthogonal. Since Q_1 and Q_2 are $n \times n$, $Q_{1\&2}$ is $2n \times 2n$.

Step 3. This step updates the northeastern quadrants of N_e and S_e , and so computes $n/2 \times n/2$ matrices U_n and U_s :

$$\begin{bmatrix} U_n \\ U_s \end{bmatrix} = Q_1^T \begin{bmatrix} N_e \downarrow \mathbf{ne} \\ S_e \downarrow \mathbf{ne} \end{bmatrix}.$$

Multiplying both sides by Q_1 ,

$$Q_1 \begin{bmatrix} U_n \\ U_s \end{bmatrix} = Q_1 Q_1^T \begin{bmatrix} N_e \downarrow \mathbf{ne} \\ S_e \downarrow \mathbf{ne} \end{bmatrix} = \begin{bmatrix} N_e \downarrow \mathbf{ne} \\ S_e \downarrow \mathbf{ne} \end{bmatrix} \quad \text{by Lemma A.5.}$$

Step 4. Since U_s is an $n/2 \times n/2$ matrix and since $\langle Q_4, R_4 \rangle = f(U_s)$, the induction hypothesis says that Q_4 is an $n \times n$ orthogonal matrix, R_4 is an $n/2 \times n/2$ upper-triangular matrix, and $U_s = Q_4 R_4$.

Step 5. The matrix Q_5 is defined as follows:

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

Since Q_4 is orthogonal from Step 4, the second factor of this product is orthogonal from Lemma A.3. The naming of $Q_{1\&2}$ establishes that $Q_{1\&2}$ is orthogonal. So, as the product of orthogonal matrices, Q_5 is orthogonal by Lemma A.2. Since Q_4 is $n/2 \times n/2$ from Step 4 and from its naming $Q_{1\&2}$ is $2n \times 2n$, the matrix Q_5 is $2n \times 2n$.

Step 6. From Step 2, the matrix \tilde{N}_2 is an $n/2 \times n/2$ upper-triangular matrix, and from Step 4, R_4 is an $n/2 \times n/2$ upper-triangular matrix. This step computes $\langle Q_6, \tilde{N}_6 \rangle = e(\tilde{N}_2, R_4)$. So, by the induction hypothesis, Q_6 is an $n \times n$ orthogonal matrix, \tilde{N}_6 is an $n/2 \times n/2$ upper-triangular matrix, and

$$\begin{bmatrix} \tilde{N}_2 \\ R_4 \end{bmatrix} = Q_6 \begin{bmatrix} \tilde{N}_6 \\ Z \end{bmatrix}.$$

Step 7. This step defines Q_e :

$$Q_e = Q_5 \begin{bmatrix} I & Z & Z & Z \\ Z & Q_6 \downarrow \text{nw} & Q_6 \downarrow \text{ne} & Z \\ Z & Q_6 \downarrow \text{sw} & Q_6 \downarrow \text{se} & Z \\ Z & Z & Z & I \end{bmatrix}$$

From Step 6, Q_6 is orthogonal; so by Lemma A.3, the padded matrix involving Q_6 is also orthogonal. Q_5 is orthogonal from Step 5. So Q_e , as the product of orthogonal matrices, is itself orthogonal by Lemma A.2. The matrix Q_5 is $2n \times 2n$ from Step 5; Q_6 is $n \times n$ from Step 6, so its padded version is $2n \times 2n$; thus Q_e is $2n \times 2n$.

Naming of \tilde{N}_e . This step names \tilde{N}_e to be

$$\tilde{N}_e = \begin{bmatrix} \tilde{N}_1 & U_n \\ Z & \tilde{N}_6 \end{bmatrix}$$

The matrices \tilde{N}_1 and \tilde{N}_6 are $n/2 \times n/2$ upper-triangular matrices from Steps 1 and 6, respectively. Since $\tilde{N}_e \downarrow \mathbf{sw} = Z$, the matrix \tilde{N}_e is an $n \times n$ upper-triangular matrix.

Step 7 establishes that Q_e is a $2n \times 2n$ orthogonal matrix. The naming of \tilde{N}_e establishes that \tilde{N}_e is an $n \times n$ upper-triangular matrix.

It just remains to show the computational correctness of e :

$$\begin{aligned}
Q_e \begin{bmatrix} \tilde{N}_e \\ Z \end{bmatrix} &= Q_5 \begin{bmatrix} I & Z & Z & Z \\ Z & Q_6 \downarrow \text{nw} & Q_6 \downarrow \text{ne} & Z \\ Z & Q_6 \downarrow \text{sw} & Q_6 \downarrow \text{se} & Z \\ Z & Z & Z & I \end{bmatrix} \begin{bmatrix} \tilde{N}_1 & U_n \\ Z & \tilde{N}_6 \\ Z & Z \\ Z & Z \end{bmatrix} && \text{by Step 7} \\
&= Q_5 \begin{bmatrix} \tilde{N}_1 & U_n \\ Z & \tilde{N}_2 \\ Z & R_4 \\ Z & Z \end{bmatrix} && \text{by Step 6} \\
&= Q_{1\&2} \begin{bmatrix} I & Z & Z & Z \\ Z & I & Z & Z \\ Z & Z & Q_4 & Z \\ Z & Z & Z & I \end{bmatrix} \begin{bmatrix} \tilde{N}_1 & U_n \\ Z & \tilde{N}_2 \\ Z & R_4 \\ Z & Z \end{bmatrix} && \text{by Step 5} \\
&= Q_{1\&2} \begin{bmatrix} \tilde{N}_1 & U_n \\ Z & \tilde{N}_2 \\ Z & Q_4 R_4 \\ Z & Z \end{bmatrix} = Q_{1\&2} \begin{bmatrix} \tilde{N}_1 & U_n \\ Z & \tilde{N}_2 \\ Z & U_s \\ Z & Z \end{bmatrix} && \text{by Step 4.}
\end{aligned}$$

Continuing the derivation:

$$\begin{aligned}
Q_{1\&2} \begin{bmatrix} \tilde{N}_1 & U_n \\ Z & \tilde{N}_2 \\ Z & U_s \\ Z & Z \end{bmatrix} &= \begin{bmatrix} Q_1 \downarrow \text{nw} & Z & Q_1 \downarrow \text{ne} & Z \\ Z & Q_2 \downarrow \text{nw} & Z & Q_2 \downarrow \text{ne} \\ Q_1 \downarrow \text{sw} & Z & Q_1 \downarrow \text{se} & Z \\ Z & Q_2 \downarrow \text{sw} & Z & Q_2 \downarrow \text{se} \end{bmatrix} \begin{bmatrix} \tilde{N}_1 & U_n \\ Z & \tilde{N}_2 \\ Z & U_s \\ Z & Z \end{bmatrix} \\
&= \begin{bmatrix} Q_1 \downarrow \text{nw} \tilde{N}_1 & Q_1 \downarrow \text{nw} U_n + Q_1 \downarrow \text{ne} U_s \\ Z & Q_2 \downarrow \text{nw} \tilde{N}_2 \\ Q_1 \downarrow \text{sw} \tilde{N}_1 & Q_1 \downarrow \text{sw} U_n + Q_1 \downarrow \text{se} U_s \\ Z & Q_2 \downarrow \text{sw} \tilde{N}_2 \end{bmatrix} \\
&= \begin{bmatrix} Q_1 \downarrow \text{nw} \tilde{N}_1 & N_e \downarrow \text{ne} \\ Z & Q_2 \downarrow \text{nw} \tilde{N}_2 \\ Q_1 \downarrow \text{sw} \tilde{N}_1 & S_e \downarrow \text{ne} \\ Z & Q_2 \downarrow \text{sw} \tilde{N}_2 \end{bmatrix} \quad \text{by Step 3} \\
&= \begin{bmatrix} N_e \downarrow \text{nw} & N_e \downarrow \text{ne} \\ Z & N_e \downarrow \text{se} \\ S_e \downarrow \text{nw} & S_e \downarrow \text{ne} \\ Z & S_e \downarrow \text{se} \end{bmatrix} \quad \text{by Steps 1 and 2} \\
&= \begin{bmatrix} N_e \\ S_e \end{bmatrix}.
\end{aligned}$$

The function e does, in fact, satisfy its computational postcondition.

So, the induction case of e is proven for all of its postconditions. \square

A.4 QR Factorization Success

While the previous section proves the algebraic correctness of the algorithm, it is also necessary to prove that the algorithm preserves certain matrix properties, non-singular being the most important.

One important attribute of a matrix is its *determinant* whose definition (see Golub and Van Loan [24, Section 2.1.4]) is not nearly as important as its properties:

Lemma A.6 *Let A , B , and Q be $n \times n$ matrices; suppose that Q is orthogonal. Then,*

$$\begin{aligned}\det(AB) &= \det(A) \det(B); \\ \det(Q) &= \pm 1; \\ \det(A) \neq 0 &\text{ iff } A \text{ is nonsingular.}\end{aligned}$$

Proof See Golub and Van Loan [24, Section 2.1.4] and Friedberg et al. [21, pp. 209–211, 340]. □

These properties result in this theorem:

Theorem A.1 *Let A be an $n \times n$ matrix. Let $\langle Q, R \rangle = f(A)$ as defined in Figure 5.2. Then $\det(A) = \pm \det(R)$.*

Proof Since Q is orthogonal and by Lemma A.6,

$$\det(A) = \det(QR) = \det(Q) \det(R) = \pm \det(R). \quad \square$$

This theorem makes Theorem 5.2 from Section 5.4.1 easy to prove:

Theorem 5.2 *Let A be an $n \times n$ matrix. Let $\langle Q, R \rangle = f(A)$ as defined in Figure 5.2. Then R is singular if and only if A is singular; Q is always non-singular.*

This theorem (technically, a corollary of Theorem A.1) follows easily from Theorem A.1 and from Lemma A.6 and because all orthogonal matrices have an inverse (by definition and Lemma A.5).

Other factorizations (like LU factorization) can run into problems when submatrices of A are singular. This is not the case with QR factorization since the orthogonal Givens rotations preserve the non-singularity of the whole matrix. The Givens rotations do all of the pivoting implicitly that must be done explicitly in a factorization like LU factorization.

Bibliography

- [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.
- [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, November 1990.
- [3] T. Axford. *Advances in Parallel Algorithms*, chapter 2, pages 26–65. In Kronsjö and Shumsheruddin [31], 1992.
- [4] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors. *Templates for the Solution of Algebraic Eigenvalue Problems*. SIAM, Philadelphia, 2000.
- [5] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, editors. *Templates for the*

- Solution of Linear Systems: Building Blocks for Iterative Methods*, chapter 4.3, pages 57–67. SIAM, Philadelphia, 1994.
- [6] P. H. Beckman. *Parallel LU Decomposition for Sparse Matrices Using Questrees on a Shared-Heap Multiprocessor*. Ph.D. dissertation, Indiana University, Computer Science Department, May 1993.
- [7] 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.
- [8] R. Bird. *Introduction to Functional Programming using Haskell*, second edition. Prentice Hall Europe, Essex, England, 1998.
- [9] 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.
- [10] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, January 1977.
- [11] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottentodi. Nonlinear array layouts for hierarchical memory systems. In *Proc. '99 Int. Conf. on Supercomputing*, pages 444–453. ACM Press, New York, June 1999.
- [12] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottentodi. Recursive array layouts and fast parallel matrix multiplication. In *Proc. 11th ACM Symp.*

- on Parallel Algorithms and Architectures*, pages 222–231. ACM Press, New York, 1999.
- [13] M. Cole. *Advances in Parallel Algorithms*, chapter 1, pages 1–25. In Kronsjö and Shumsheruddin [31], 1992.
- [14] 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, November 1996.
- [15] J. Dongarra. *Templates for the Solution of Algebraic Eigenvalue Problems*, pages 315–319. In Bai et al. [4], 2000.
- [16] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990.
- [17] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, March 1988.
- [18] 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.
- [19] P. C. Fischer and R. L. Probert. Storage reorganization techniques for matrix computation in a paging environment. *Commun. ACM*, 22(7):405–415, July 1979.

- [20] 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.
- [21] S. H. Friedberg, A. J. Insel, and L. E. Spence. *Linear Algebra*, second edition. Prentice-Hall, New York, 1989.
- [22] 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, October 1999.
- [23] G. H. Golub, R. J. Plemmons, and A. Sameh. Parallel block schemes for large-scale least-squares computations. In *High-Speed Computing, Scientific Applications and Algorithm Design*, pages 171–179. University of Illinois Press, Urbana-Champaign, 1988.
- [24] G. H. Golub and C. F. Van Loan. *Matrix Computations*, third edition. The John Hopkins University Press, Baltimore, 1996.
- [25] F. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Develop.*, 41(6):737–755, November 1997.
- [26] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, 1996.
- [27] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, July 1961.
- [28] 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.
- [29] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, third edition, volume 1 of *The Art of Computer Programming*. Addison Wesley Longman, Boston, New York, 1997.
- [30] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, second edition, volume 3 of *The Art of Computer Programming*. Addison Wesley Longman, Boston, New York, 1998.
- [31] L. Kronsjö and D. Shumsheruddin, editors. *Advances in Parallel Algorithms*. John Wiley & Sons, New York, 1992.
- [32] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.
- [33] H. R. Lewis and L. Denenberg. *Data Structures & Their Algorithms*, third edition. HarperCollins, New York, 1991.
- [34] A. C. McKellar and E. G. Goffman, Jr. Organizing matrices and matrix operations for paged-memory systems. *Commun. ACM*, 12(3):153–165, March 1969.
- [35] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Ottawa, Ontario: IBM Ltd., March 1966.
- [36] S. S. Muchnick. *Advanced Computer Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

- [37] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*, second edition. Morgan Kaufmann, San Francisco, 1998.
- [38] F. Rabhi and G. Lapalme. *Algorithms: A Functional Programming Approach*. Addison-Wesley, Boston, 1999.
- [39] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.
- [40] Silicon Graphics, Inc. R8000 microprocessor chip set. Technical report, Silicon Graphics, Inc., 1994.
- [41] J. Spieß. Untersuchungen des zeitgewinns durch neue algorithmen zur matrix-multiplikation. *Computing*, 17(1):23–36, 1976.
- [42] G. L. Steele, Jr. Debunking the “expensive procedure call” myth, or procedure call implementations considered harmful, or LAMBDA: the ultimate GOTO. *ACM77: Proc. 1977 Ann. Conference*, pages 153–162, 1977.
- [43] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.
- [44] V. Valsalam and A. Sjkellum. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concur. Comput. Prac. Exper.*, page in press.
- [45] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proc. '98 Int. Conf. on Supercomputing*. ACM Press, New York, 1998.

- [46] D. S. Wise. Undulant block elimination and integer-preserving matrix inversion. *Sci. Comp. Program.*, 33(1):29–85, January 1999.
- [47] 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.
- [48] 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.
- [49] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.
- [50] A. Y. H. Zomaya, editor. *Parallel & Distributed Computing Handbook*. Computer Engineering. McGraw-Hill, New York, 1996.

Vita

Jeremy David Frens was born in Chino, California on March 7, 1970, but he lived in New Jersey, then Wisconsin, and then Illinois before attending college. He received his B.A. from Calvin College in 1992, a double major in Computer Science and Mathematics. At Calvin College, he was awarded the William Rinck Memorial Prize in Mathematics. He received his M.S. in Computer Science from Indiana University in 1994. In 1998, he received the Outstanding Associate Instructor Award from the Computer Science Department at Indiana University.

He taught for two years as Assistant Professor at Northwestern College in Orange City, Iowa. In 2002, he took a position as Assistant Professor at Calvin College in Grand Rapids, Michigan.