

# Auto-Blocking Matrix-Multiplication or Tracking BLAS3 Performance from Source Code\*

Jeremy D. Frens and David S. Wise

Computer Science Dept., Indiana University  
Bloomington, Indiana 47405-4101, USA  
jdfrens, dswise@cs.indiana.edu

## Abstract

An elementary, machine-independent, recursive algorithm for matrix multiplication  $C+=A*B$  provides implicit blocking at every level of the memory hierarchy and tests out faster than classically optimal code, tracking hand-coded BLAS3 routines. Proof of concept is demonstrated by racing the in-place algorithm against manufacturer's hand-tuned BLAS3 routines; it can win.

The recursive code bifurcates naturally at the top level into independent block-oriented processes, that each writes to a disjoint and contiguous region of memory. Experience has shown that the indexing vastly improves the patterns of memory access at all levels of the memory hierarchy, independently of the sizes of caches or pages and without *ad hoc* programming. It also exposed a weakness in SGI's C compilers that merrily unroll loops for the super-scalar R8000 processor, but do not analogously unfold the base cases of the most elementary recursions. Such deficiencies might deter future programmers from using this rich class of recursive algorithms.

## Categories and subject descriptors:

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

**General Term:** Performance.

**Additional Key Words and Phrases:** storage management, indexing, quadtrees, swapping, cache misses, paging.

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

© 1997 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

*Proc. 1997 ACM Symp. on Principles and Practice of Parallel Programming, SIGPLAN Notices* **32**, 7 (July 1997) 206–216.

## 1 Introduction

This paper revisits matrix algebra, specifically multiplication, to explore an algorithm nearly as simple as traditional ones, and certainly better for hierarchical memory. We present a simple recursive algorithm and a matrix representation suited to it that have outperformed hand-optimized BLAS3 matrix multiplication [10]. Hand-coded by the manufacturer for salesman's performance claims, this BLAS3 multiplication is widely thought to offer optimal performance.

Experiments with the algorithm have exposed weaknesses in production compilers: loops are unrolled but recursions are not unfolded. Moreover, they demonstrate a new way to balance the parallel schedules top-down, following the recursion and partitioning the matrices for favorable run-time locality.

It is well known that linear systems are best solved by algorithms that decompose matrices into blocks [9, 10]. This paper focuses on an elementary, machine-independent, recursive algorithm for matrix multiplication  $C+=A*B$  that provides implicit blocking at every level of the memory hierarchy and tests out faster than classically optimal code. Proof of concept is demonstrated by racing it against hand-coded BLAS3 routines, which lose to it as they are forced into paging.

With memory hierarchies of the future layered to still deeper levels, in-place algorithms sensitive to the hierarchy will be necessary to solve large problems. Insensitive to the exact quantum of memory-transfer at any level, our new algorithm retains its efficiency as it decomposes and solves independent problems that cross through cache, main RAM, paged, and distributed levels of memory. Although additional storage may be available at distributed processors, it uses none, except for local variables in a recursion stack of depth  $\lg n$  for order  $n$  matrices—constant space for all practical purposes.

Four insights underlie the algorithms. First is the quadtree decomposition of matrices [23], as well as the algorithms that manipulate them using recursive descent [24]. Second is a familiar indexing, newly applied to the map of matrices onto the address space. Next is a decomposition of the usual eight recursive, quadrant multiplications into two parallel streams, balancing computational loads when the factors have known padding (east or south) with zeroes. The last insight is a careful dovetailing of successive calls that has the effect, at a low level, of reusing data already resident in each level of memory and, at a high level, of balancing the computational load among sibling processes.

### 1.1 Whence blocking?

The reader should distinguish our introduction of blocking via recursive data structures and programs, from compile-time transfor-

mations of conventional arrays and loops to obtain good blocking at run time. The latter strategy uses well the compiler’s knowledge of the hardware parameters to fit the target machine, but is also limited to patterns of source code and transformations anticipated by the compiler writer. Recent work of this sort includes data transformations between row- and column-major [1, 6], blocking of high-level code specifically to admit lower-level BLAS3 invocations [2], and more recently such blocking to bypass BLAS3 entirely [5].

One could interpret our representations and algorithms as a new way to introduce blocking into existing programs. We prefer to view them, instead, as results from a different style of expressing the same high-level algorithms. The style is, however, one that also inspires a new perspective, new insights, and—perhaps also—new algorithms. Recursive and parallel versions of practical algorithms already exist that use this structure well [24, 13]; in such contexts this matrix multiplication is not only the most natural, it is also the fastest. Importantly, no conversions between data representations are necessary there (aside from the usual ones at input and output.)

Fateman’s recent treatise that, in part, distinguishes matrices from arrays [11, §4.3] is relevant here. Matrix problems deal with underlying vector spaces that are better decomposed top-down into subspaces. While quadtree decomposition may not be the most efficient one for any given problem, it does enforce the divide-and-conquer perspective that algebra allows us. Viewing the matrix as decomposed by rows or by columns is a bottom-up approach; subspaces may still be visible, but they assemble themselves differently.

Our philosophy derives from our experience with functional programming and our view of its critical role for parallel programming [4]. Linear systems, in particular, come to us with a rich algebra that is best visible in a functional program that reads like interdependent formulae [13, 24]. Discovery of good formulations of known—and maybe even unknown—techniques follows top-down, partition-and-conquer of the underlying vector space. Our experience is that many insights and efficiencies are to be found here.

This paper represents our effort to carry this philosophy back to serially addressed, hierarchical memory using concise and efficient source code. The C programs here should be read, first, as an idealized compilation of functional source code (like a multiplication of HASKELL arrays [16]) to run well on extant high-performance systems.

## 1.2 Outline of paper.

The remainder of this paper is in six parts. A short section reviews the conventional looping algorithms that multiply using inner-, outer-, or middle- products, and is followed by Section 3 giving the definitions of the quadtree representation of matrices and the indexing on it. The next section contains details on the quadtree recursions studied here, and Section 5 presents experimental results. Section 6 offers the seminal analysis that only two blocks need be reloaded between any block multiplications and that square blocks—like these quadrants—are optimal. The last section offers conclusions.

## 2 Classic loops

The algorithms in this paper are presented in C. The benchmark code for matrix-matrix multiplication,  $C+=A*B$ , is the conventional inner-product code of Figure 1, as presented in most linear algebra courses. Following the associativity of addition, the three nested-loop controls can be permuted [14, p. 19] and reordered to obtain outer- and middle-product alternatives. All exhibit the same problem: that elements of some of the arrays must be fetched into faster

```
void loop_multiply (int order, Scalar *c, Scalar *a, Scalar *b) {
  /* Assert that matrix c has already been zeroed. */
  for (register int i = 0; i < order; i++)
    for (register int j = 0; j < order; j++)
      for (register int k = 0; k < order; k++)
        c[i + j*order] += a[i + k*order] * b[k + j*order];
}
```

Figure 1: Matrix multiplication with three nested loops.

```
void loop_multiply (int order, Scalar *c, Scalar *a, Scalar *b) {
  for (register int i = 0; i < order; i++)
    for (register int j = 0; j < order; j++) {
      accumulator = 0;
      for (register int k = 0; k < order; k++)
        accumulator += a[k + i*order] * b[k + j*order];
      c[i + j*order] = accumulator;
    }
}
```

Figure 2: Three nested loops with zeroing on pre-transposed A.

memory unfortunately often. (See Section 6.)

Several transformations of this code are common. One initializes the matrix C as it goes. Another transposes the default representation of column-major order to represent the A array, instead, in row-major order to avoid excessive cache misses on serially reading from A in the inner loop. Both of these are illustrated in Figure 2.

It is usual to modify this code, particularly within BLAS3 codes, so that the nested multiply-add (or `saxpy`) applies to blocks, rather than to scalars. The size of the block is selected to fit the machine’s register and cache capacities, so the resulting code is not portable.

The ubiquity of such code, moreover, leads optimizing compilers to unroll inner loops in order to avoid excess testing, to fill an instruction pipe, and to take full advantage of the capacity of the instruction cache. Silicon Graphics’ C compiler for its MIPS R8000 chip, for instance, automatically unrolls this inner loop twice as inline code to take advantage of its super-scalar processing.

## 3 Quadtree decomposition of matrices

On first reading of the definitions for the quadtree decomposition of matrices, it is easier to assume that the order of the matrix is a power of two. This restriction is relaxed later with negligible overhead; the timing curves are smooth.

**Definition 1** [23] *A complete matrix has index 0. A matrix at index  $i$  is either scalar, or it is composed of four submatrices—northwest, southwest, southeast, and northeast—each of half the order and with indices  $4i+1$ ,  $4i+2$ ,  $4i+3$ , and  $4i+4$ , respectively.*

Figure 3 presents a context of C declarations for developing the quadtree-matrix codes. The constant `offset` specifies the number of interior nodes (e.g. 5 in Figure 4). It can also be computed

```
typedef float Scalar;
typedef struct {
  int order;           /* Size of the matrix */
  int offset;         /* Census of nonterminal nodes */
  Scalar *matrix;
} Matrix;
#define nw(i) ((i << 2) + 1) /* index to Northwest quadrant of Matrix i */
#define sw(i) ((i << 2) + 2) /* index to Southwest quadrant of Matrix i */
#define se(i) ((i << 2) + 3) /* index to Southeast quadrant of Matrix i */
#define ne(i) ((i << 2) + 4) /* index to Northeast quadrant of Matrix i */
```

Figure 3: An environment for matrix representation

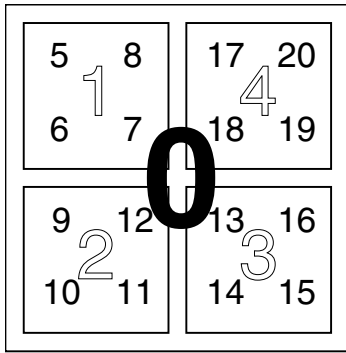


Figure 4: Level-order indexing of blocks in matrix of order 4.

directly from order:

$$\text{offset} = \sum_{i=0}^{\lceil \lg(\text{order}) \rceil} 4^i = \frac{4^{\lceil \lg(\text{order}) \rceil} - 1}{3}.$$

Subtracting it from each element's actual index yields a zero-based indexing of just the elements in the matrix. Ignoring the offset, this definition provides a zero-based, level-order indexing<sup>1</sup> across a matrix's tree [18, p. 350, 401], as illustrated in Figure 4 [7, 3]. It prompts several observations.

- Every block/subtree is indexed consecutively at each of its levels. The scalars in any subblock, as terminal nodes at the same level in some subtree, are therefore indexed consecutively. A good, block-oriented algorithm, one that descends the tree to a subtree small enough to fit in cache (for instance), experiences excellent caching behavior without any programmers' worries over striding. Moreover, a multilayered memory experiences good locality simultaneously at each level of the memory/tree, independently of the page sizes.
- Where a matrix's order is not a power of two, it is padded (as if with zeroes) on its south and east margins to the next larger power, *but* these extra elements would never be touched by a good algorithm. (Interior padding is also possible.) Padding introduces gaps into the level-order indexing, but the data in these gaps are never touched and never migrate to cache.
- Padding can be detected from the size of each quadrant, or a block purely of zeroes (padding) might be announced by a flag in its parent node (as in Figure 8), from which all memory accesses into that quadrant become unnecessary. That flag directs the algebra around zeroes (as additive identities and multiplicative annihilators), to yield accelerated computations on sparse matrices. A good use for a second flag is to announce, in contrast, that the subtree is practically dense—that it contains no zero blocks large enough to justify the routine zero tests that might avoid them. Instead, all matrix operations would descend blindly to the leaves, saving the few stalls that would result from branch instructions.

<sup>1</sup>This indexing, like floating-point numbers and even the quadtree representation, itself, is an internal representation that speeds computation. All three are isomorphic to alternative representations that are more easily read by humans, but translations between them often are computationally difficult. In all three cases, such translations never occur during routine computation, however, and are overlapped with trudging, linear-time input/output whenever necessary.

```
int offset;
Scalar *A_matrix, *B_matrix, *C_matrix;

void naive_multiply (Matrix a, Matrix b, Matrix c) {
    offset = a.offset;
    A_matrix = a.matrix;
    B_matrix = b.matrix;
    C_matrix = c.matrix;
    mult (1, 1, 1);
}

static void mult (register Index i_C,
                 register Index i_A, register Index i_B) {
    if (i_A >= offset)
        C_matrix[i_C] += A_matrix[i_A] * B_matrix[i_B];
    else {
        mult (nw (i_C),   nw (i_A)   , nw (i_B));
        mult (se (i_C),   sw (i_A)   , ne (i_B));
        mult (nw (i_C),   ne (i_A)   , sw (i_B));
        mult (ne (i_C),   nw (i_A)   , ne (i_B));
        mult (se (i_C),   se (i_A)   , se (i_B));
        mult (sw (i_C),   sw (i_A)   , nw (i_B));
        mult (ne (i_C),   ne (i_A)   , se (i_B));
        mult (sw (i_C),   se (i_A)   , sw (i_B));
    }
}
```

Figure 5: Quadtree matrix multiplication (with a strange sequencing).

- Often the interior nodes of the tree are elided. The only represented nodes become its scalars (leaves) indexed across the terminal level. Subtracting the `offset` (5 in Figure 4) yields the block indexing that is the closest analog of the classic indexing of column-major matrices within a linear array.
- As little as a 4% space overhead (beyond the elements) allows for a byte—two flags per subtree—at every non-terminal node. For instance, the scalars in an order- $n$  `double` matrix occupy  $8m^2$  bytes (for  $m = 2^{\lceil \lg n \rceil}$ ), but its quadtree has at most  $\frac{m^2-1}{3}$  interior nodes. In such a case, the interior and exterior nodes can be allocated with contiguous indices, but in separate memory partitions.

## 4 Recursive matrix multiplication

The algorithms in this section are the same as those sketched above, except that the ordering of elementwise multiply-adds is vastly permuted. Taking advantage of recursion, the order of these operations is rearranged blockwise, for blocks coinciding with subtrees at all levels in the quadtree decomposition.

Like the algorithms above, all are written as higher-level code, and compiled with full optimization for the experiments reported in the next section. All this is generic code that contrasts with the competing BLAS3 routines that have been polished over decades to maximal performance on each architecture.

A functional programmer's model for matrix multiplication uses mapping functions over quadruples [24, 23] to decompose matrix problems into square blocks. It is not too different from Figure 5 which exposes the blockwise algorithm, but obfuscates both the functional syntax and the sparse-matrix algebra that motivated this representation. As observed elsewhere [23, 21], the quadrants of the answer can provide a partitioning into 4, 16, 64, ... processes to compute the answer independently of one another.

#### 4.1 Balanced parallel multiplication

Matrix multiplication can be balanced across a small pool of processors top-down, even at compile time if the order is known. (Then a change of order requires recompilation to a new schedule.) The following cases explain how to derive—top-down—a balanced schedule for run-time parallelism as long as one more processor remains to be scheduled. It is formulated for square matrices, as arise (*e.g.*) in matrix decomposition. Balancing is only important near the top of the scheduling tree, but it is very important there; the difficult case occurs where the order is not equal to (especially, just under) a power of two. For all orders the processing load can be balanced by halving the processor resource as the recursion descends the quadtree.

**Definition 2** A stripe<sup>2</sup> is a set of adjacent rows in a matrix. A colonnade is a set of adjacent columns.

**Definition 3** A matrix or portion thereof is full if it has no known/significant internal zeroes and, if square, no zero padding on its south and east.

The stripes and colonnades of interest here are full across the matrix. (They identify the candidate subspaces of the underlying vector space.) A special case occurs where a north stripe and a west colonnade (here necessarily the same size) intersect to form a northwest *square*. This case is distinguished and so labeled, so in this context stripes and colonnades span the matrix with non-zero elements, and squares land northwest-justified, filling the matrix only as necessary to conform to a stripe as left factor or a colonnade as right factor.

0. **Balanced Square.** When two full matrices are multiplied, eight balanced quadrant multiplications arise; they can elegantly be partitioned among four independent parallel threads—one for each quadrant of the product,  $C$ . This case, illustrated at the top of Figure 6, is called a *balanced square*. We split it, instead, into only two parallel threads (*e.g.* the east and west colonnades of  $C$ ) to simplify process management; other cases, below, bifurcate similarly. If a third or fourth processor were available, then it would be committed in another bifurcation at the next level of the quadtree.

If we would allow temporary storage to the algorithm, then this is the case where Strassen's recurrence [21, 15, §1.3.8] applies best (except on sparse or tiny matrices).

Otherwise, there are seven cases where padding might unbalance a top-down parallel dispatch, classified by the bottom sketches in Figure 6. All of them have two subcases, depending on whether the padded dimension(s) constrain nonzero entries to the north and/or west quadrants, or whether they wash into the south and/or east. In the former case, four (or more) of the quadrant multiplications are annihilated. In the latter case all eight proceed, including a northwest contribution that is, entirely, balanced squares.

In order to balance the bifurcations, Case 2a is delegated to one branch, with its paired Case 2b going to the other. Similarly, Case 3a and Case 3b are assigned opposing branches to balance one, another. The more numerous Cases 0, 4, and 5 are all scheduled serially, anticipating their bifurcations at the next level of the tree. Cases 1, 2, and 3 occur less frequently than the others and the bother for their balancing might at first appear silly. However, it becomes critical nearer the root of the quadtree (where, after all, processing resources will yet be more plentiful) to correct gross imbalances that can be introduced by heavy padding.

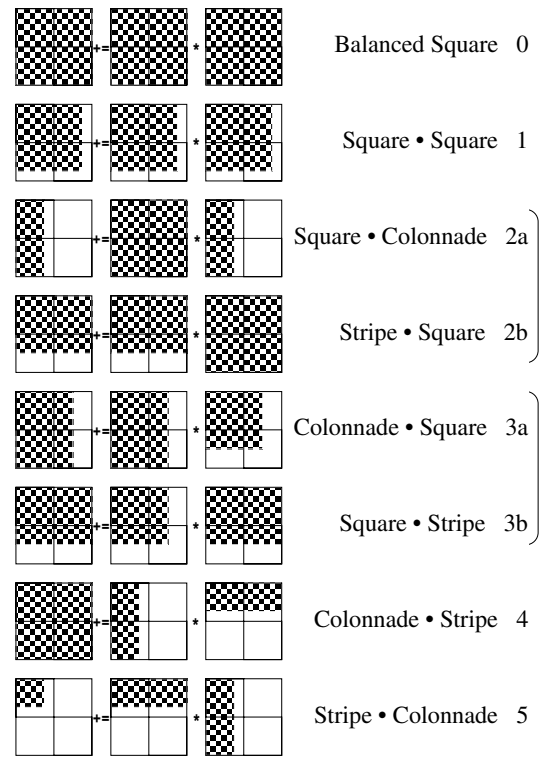


Figure 6: Classification of how south and east padding unbalances quadrant multiplications.

1. **Square•Square.** In the event that the square is entirely in the northwest quadrant, a single recursion results—including the possibility of a balanced square (Case 0) when it fits there (almost) exactly.

Otherwise (when it extends into the southeast), cleaving a square matrix into quadrants results in a full northwest quadrant, a full stripe as the southwest quadrant, another square to the southeast, and a full colonnade to the northeast. The size of the stripe and colonnade conform and, so, can be used for *a priori* balancing. This case reduces to eight function calls, one of each of these eight cases. Cases 0, 1, 4, and 5 are done serially, but the four conforming halves of Cases 2 and 3 are scheduled as two balanced threads.

2. The following two cases generate equivalent computational load, and so are gathered.

(a) **Square•Colonnade.** The result  $C$  can be partitioned north–south yielding two balanced threads.

If the colonnade is purely western, then there are four nontrivial block-multiplications, each Case 2a unless the west is full (Case 0). In either case, they can be balanced across two independent threads. If the colonnade washes into the east, then the four cases just discussed are still needed, but there are four additional, balanced Case 0 problems on the west half of  $C$ .

(b) **Stripe•Square.** This situation is analogous to Case 2a. The result  $C$  can be partitioned east–west yielding two balanced threads.

<sup>2</sup>as in “Stars and Stripes,” with apologies to tigers and zebras.

If the full stripe doesn't paint into the southern half of the matrix, then there are only four subproblems, all Case 2b or possibly 0, which can be balanced east-west. Otherwise, there are four more Case-0 multiplications in the north.

3. The next two cases also generate the same computational load.
  - (a) **Colonnade•Square.** The result  $C$  can be partitioned north-south yielding two balanced threads. If the colonnade is all in the west, then there are only two balanced Case 3a recurrences (maybe Case 0), which can be scheduled in parallel. Otherwise there are two each of Case 0, 2a, 3a, and 4 that can form two balanced, independent threads.
  - (b) **Square•Stripe.** The result  $C$  can be partitioned east-west yielding two balanced threads. If the stripe is purely northern, then there are only two independent Cases 3b (perhaps 0). Otherwise there are two each of Case 0, 2b, 3b, and 4 that may form two independent threads.
4. **Colonnade•Stripe.** This case augments the most elements in the product for each non-zero element in the factors; the extreme of this case is the outer product of two vectors. The result  $C$  can be partitioned north-south yielding two balanced threads.

If the colonnade is western only, then there are four independent, direct recursive calls that are balanced for parallelism. If it washes into the east, there are eight recursive calls again: four of Case 0 and four of Case 3a that also bifurcate north-south and balance.

5. **Stripe•Colonnade.** This case has the fewest elements augmented by the multiplication for each non-zero element in the factors; the extreme is the inner product of two vectors.

If the colonnade is western only, then no new threads are spawned; both recurrences are done serially. If it washes into the east, there are eight recursive calls again: two Case 0 serially, two of Case 5 serially, and two each of Case 2a and 2b that are serial but pairwise balanced and they can be scheduled in parallel.

All the south and east zeroes here are presumed to be padding on the matrix, raising its physical order to a power of two. As a result, Cases 1-7 are expected to be far rarer than Case 0. They arise only from  $O(n)$  blocks on the south and east perimeter of a matrix, but Case 0 applies on the  $O(n^2)$  blocks in its core. Even with balancing, Case 0 arises repeatedly in all eight recurrences.

Testing for padding is run-time overhead, as suggested by the extra base case in Figure 8. Similar tests might also handle sparse matrices, but the extra tests take marginally extra cycles and can be elided when the factors are known to be dense and full.

Classification on the order of a multiplication should not be too precise here. For instance, a square multiplication of order 1023 should be treated as 1024 (Case 0), because that case admits more parallelism at a higher level in the tree (for fewer dispatches) than Case 1 would, and because the cost of repeated testing as the tree is descended becomes uneconomical here. Carrying out the scalar products on its padding of 2047 zero elements (using simply the Figure-7 code) has proven to be cheaper than the run-time tests at its leaves to short circuit the annihilators.

## 4.2 Spinning code for cache reuse

Although the quadrant-wise recursion of Figure 5 offers parallelism and balancing, it lacks a feature exposed analytically—that only two of the three operands need be reloaded between one step and the next. While such a quadrant recursion had been familiar to us before that analysis, this important attribute and its realization in Figure 7 is new.

A split of the recursive algorithm into two versions was necessary to realize this property. Labeling of the two as `up_multiplication` and `dn_multiplication` suggests that they are duals, like LEFT and RIGHT; the terms UP and DOWN are used because they won't get overloaded. The assertions in Figure 7's comments form an inductive proof that only two of the three blocks (from  $A$ ,  $B$ ,  $C$ ) need be reloaded across any semicolon, including bridging from each block-exit to the succeeding block-entry. One of the three always remains resident in the primary cache, or secondary cache, or RAM (when paging) between any two steps, regardless of the size of that layer of physical memory.

Blocks from  $A$  and  $B$  are read-only, of course. Blocks from  $C$  become dirty as they are over-written but, if threads are balanced top-down (as in Section 4.1), no other processor will target the same subblock of  $C$  and suffer cache-incoherency from another's "dirt." Of the eight quadrant operations, therefore, those on each quadrant of writable operand  $C$  occur consecutively, so as to emit both updates while it is yet cache-resident. Quadrants from  $A$  and  $B$ , which are swapped more often, are read-only. Extra compound statements are marked by otherwise superfluous braces in Figure 7 to suggest how it bifurcates into independent, parallel threads.

This is the skeleton of the algorithm whose caching and swapping is tested in the next section. It has the virtue of omitting all architectural parameters that would interfere with its portability from one machine to the next. It recursively decomposes the problem until three square blocks fit nicely in cache (or pages for the square blocks fit nicely in main memory), without parameterizing the size of those blocks.

**Such cache-conservation immediately halved the running times** on a quiet system with a large cache (the SGI POWER CHALLENGE). The impact was less on another system with a small data cache (a DEC ALPHA).

Finally, we observe that this "two miss" property has intentionally been destroyed in Figure 5. Unless the compiler can reorder these function calls, all three of the operands need to be reloaded between any two steps; so, that code represents a subtle, worst case for cache reuse. We find it interesting that a relatively obscure permutation of the eight recursive calls—away from any "natural" order (for instance, one that arises from mapping functions)—was necessary to exhibit so many cache misses.

## 4.3 Unfolding for superscalar architecture

It is necessary to unfold the recursion slightly, both to take proper advantage of instruction cache, and also to enable a fair comparison with compiled C loops like Figure 1's that are routinely unrolled. Compilers for the SGI POWER CHALLENGE, for instance, will unroll inner loops twice (or more) to take advantage of its super-scalar architecture, but recursive calls are not similarly unfolded.

Management for the instruction cache ought to be handled by compilers, targeted as they are to specific hardware. Unfortunately, these compilers do a poor job with ordinary function linkage, stacking when it is unnecessary, and not unfolding recursions at all. (Nothing new there [22]!) So C's macro facility was used to unfold the base case manually (within constraints of the instruction cache, as a good compiler would). A typical result of unfolding appears in Figure 8. With the base case composed of eight in-line,

```

void multiply (Matrix a, Matrix b, Matrix c) {
    offset = a.offset;
    A_matrix = a.matrix;
    B_matrix = b.matrix;
    C_matrix = c.matrix;
    up_mult (1, 1, 1);
}

static void dn_mult (register Index i_C, register Index i_A, register Index i_B) {
    /* All assertions about cache refer to extreme corners of */
    /* the named quadrant. */
    if (i_A >= offset)
        C_matrix[i_C] += A_matrix[i_A] * B_matrix[i_B];
    else { /* decompose C,A,B into nw, ne, sw, se. */
        /* Precondition: one extreme block of C_ne,A_nw, or B_ne in cache. */
        /* Load other two of C_ne, A_nw, B_ne; */
        {{dn_mult (ne (i_C), nw (i_A), ne (i_B)); /* Leaving C_ne_nw in cache. */
        /* Load A_ne, B_ne; */
        up_mult (ne (i_C), ne (i_A), se (i_B));} /* Leaving B_se_ne in cache. */
        /* Load C_se, A_se, */
        {dn_mult (se (i_C), se (i_A), se (i_B)); /* Leaving C_se_nw in cache. */
        /* Load A_se, B_se; */
        up_mult (se (i_C), sw (i_A), ne (i_B));} /* Leaving A_sw_nw in cache. */
        /* Load C_sw, B_nw; */
        {{up_mult (sw (i_C), sw (i_A), nw (i_B)); /* Leaving C_sw_nw in cache. */
        /* Load A_se, B_sw; */
        dn_mult (sw (i_C), se (i_A), sw (i_B));} /* Leaving B_sw_ne in cache. */
        /* Load C_nw, A_ne */
        {up_mult (nw (i_C), ne (i_A), sw (i_B)); /* Leaving C_nw_nw in cache. */
        /* Load A_nw, B_nw; */
        dn_mult (nw (i_C), nw (i_A), nw (i_B));}
        /* Postcondition: extreme blocks of C_nw, A_nw, B_nw in cache. */
    }
}

static void up_mult (register Index i_C, register Index i_A, register Index i_B) {
    if (i_A >= offset)
        C_matrix[i_C] += A_matrix[i_A] * B_matrix[i_B];
    else { /* decompose A,B,C into nw, ne, sw, se. */
        /* Precondition one extreme block of C_nw,A_nw, or B_nw in cache. */
        /* Load other two of C_nw, A_nw, B_nw; */
        {{up_mult (nw (i_C), nw (i_A), nw (i_B)); /* Leaving C_nw_ne in cache. */
        /* Load A_ne, B_sw; */
        dn_mult (nw (i_C), ne (i_A), sw (i_B));} /* Leaving B_sw_nw in cache. */
        /* Load C_sw, A_se, */
        {up_mult (sw (i_C), se (i_A), sw (i_B)); /* Leaving C_sw_ne in cache. */
        /* Load A_sw, B_nw; */
        dn_mult (sw (i_C), sw (i_A), nw (i_B));} /* Leaving A_sw_nw in cache. */
        /* Load C_se, B_ne; */
        {{dn_mult (se (i_C), sw (i_A), ne (i_B)); /* Leaving C_se_nw in cache. */
        /* Load A_se, B_se; */
        up_mult (se (i_C), se (i_A), se (i_B));} /* Leaving B_se_ne in cache. */
        /* Load C_ne, A_ne */
        {dn_mult (ne (i_C), ne (i_A), se (i_B)); /* Leaving C_ne_nw in cache. */
        /* Load A_nw, B_ne; */
        up_mult (ne (i_C), nw (i_A), ne (i_B));}
        /* Postcondition: extreme blocks of C_ne, A_nw, B_ne in cache. */
    }
}

```

Figure 7: Two-miss algorithm for quadtree matrix multiply.

```

static void dn_mult (register Index i_C, register Index i_A, register Index i_B) {
    if (i_A >= adjOffset) {
        C_matrix[ne (i_C)] += A_matrix[nw (i_A)] * B_matrix[ne (i_B)];
        C_matrix[ne (i_C)] += A_matrix[ne (i_A)] * B_matrix[se (i_B)];
        C_matrix[se (i_C)] += A_matrix[se (i_A)] * B_matrix[se (i_B)];
        C_matrix[se (i_C)] += A_matrix[sw (i_A)] * B_matrix[ne (i_B)];
        C_matrix[sw (i_C)] += A_matrix[sw (i_A)] * B_matrix[nw (i_B)];
        C_matrix[sw (i_C)] += A_matrix[se (i_A)] * B_matrix[sw (i_B)];
        C_matrix[nw (i_C)] += A_matrix[ne (i_A)] * B_matrix[sw (i_B)];
        C_matrix[nw (i_C)] += A_matrix[nw (i_A)] * B_matrix[nw (i_B)];
    }
    else if ( isZero( A_decoration[i_A] ) || isZero( B_decoration[i_B] ) ) { }
    else {
        {dn_mult (ne (i_C),          nw (i_A),          ne (i_B));
         up_mult (ne (i_C),          ne (i_A),          se (i_B));
         dn_mult (se (i_C),          se (i_A),          se (i_B));
         up_mult (se (i_C),          sw (i_A),          ne (i_B));}
        {up_mult (sw (i_C),          sw (i_A),          nw (i_B));
         dn_mult (sw (i_C),          se (i_A),          sw (i_B));
         up_mult (nw (i_C),          ne (i_A),          sw (i_B));
         dn_mult (nw (i_C),          nw (i_A),          nw (i_B));}
    }
}

```

Figure 8: Typical unfolding of Figure 7’s base case.

scalar multiply-adds, a super-scalar architecture can overlap two or more of them.

We did observe **another improvement from this unfolding of almost a full factor of two** on our POWER CHALLENGE, enabling superscalar performance within the capacity of its R8000 processor.

Figure 8 also illustrates how a byte array of “decorations” on the nonterminal nodes of the quadtree can be used at run time to steer the multiplication around zero blocks. The zeroes may arise from padding or in empty reaches of sparse matrices. The new base case tests such a decoration, detecting when a product is to be annihilated instead of computed. Architectural prejudices against tests in super-scalar code and economy-of-scale both suggest that such tests are more effective near the root of the quadtree.

## 5 Experimental results

The experiments of this section were carried out principally on two machines. The first machine is an SGI ONYX with four R4400 processors, 64 megabytes of RAM and a large swapping disk. The second machine is an SGI POWER CHALLENGE with ten R8000 processors, 4 megabytes of secondary cache per processor, 2 gigabytes of shared RAM, and 6 gigabytes of shared swapping space. Both used version 6.0.1 of the manufacturer’s C compiler. The QUADTREE algorithm was also run on a DEC ALPHA AXP7720 with 1.5 gigabytes of RAM to see how it tracks on a different architecture.

Four algorithms were tested: (1) the naive inner-product form of Figure 1 (INPROD); (2) the transposed inner-product of Figure 2 (TRANSINPROD); (3) the QUADTREE algorithm of Figures 7 and 8 using the internal nodes of the quadtree to hold flags to signal zero and dense blocks; and (4) the BLAS3 routine `dgemm`.

This last routine has been hand-coded by the manufacturer and exhibits as much as 264-Mflop performance in these tests on a 300 Mflop chip. Even though this rate is not sustained across all these tests, we take it to be a credible target. Our timings arise from a not-very-polished high-level program; Figure 7 compares more closely to Figure 1 than to assembly code.

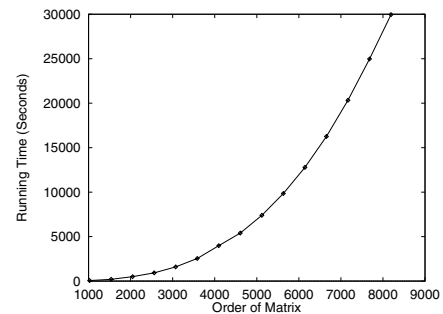


Figure 9: Uniprocessor Performance of Quadtree Matrix-Matrix Multiplication

### 5.1 Uniprocessing behavior.

Figure 9 shows the running time for the QUADTREE algorithm on the SGI POWER CHALLENGE for variously sized matrices. Despite natural affinity of the QUADTREE algorithm towards matrices whose orders are powers of two, the decoration flags direct the algorithm efficiently for matrices of all orders. The algorithm has also been tested on consecutive orders, and running times do remain smooth. As the graph shows, as the order of the matrix doubles, running times increase by a factor of eight, as predicted by familiar analysis.

Table 1 presents running times for the four matrix-matrix algorithms on the SGI Power Challenge. The times for INPROD progressively gets worse and worse as the order increases since the striding traversal of  $A$  is hardly cache sensitive. The other three algorithms are responsive to cache needs and so experience growth expected of an  $O(n^3)$  algorithm.

### 5.2 Multiprocessing behavior.

Table 2 contains the multiprocessing results for the BLAS3 and QUADTREE algorithms on the SGI ONYX and SGI POWER CHALLENGE. These times are also graphed in Figures 10 and 11. The table also displays the running time on the DEC ALPHA.

Order	BLAS3	QUADTREE	TRANSINPROD	INPROD	Ratio QUADTREE to BLAS3	Ratio TRANSINPROD to QUADTREE
1023	8.122	62.09	209.5	272.9	7.64	3.37
1025	8.120	65.21	211.1	279.4	8.03	3.24
2047	68.53	497.5	1738	3527	7.26	3.49
2049	68.41	530.3	1743	3536	7.75	3.39
3050	211.0	1675	5746	11940	7.94	3.43
4095	545.4	3978	13920	30590	7.29	3.50

Table 1: Uniprocessor running times for all four algorithms (on SGI POWER CHALLENGE)

Number of Processors	Machine	Size	QUADTREE		BLAS3		
			Time	Major faults	Time	Major faults	
1	SGI ONYX R4400	1024	180.1	0	40.4	0	
		2048	1479	1,559	374.9	1,386	
		3030	4722	21,279	1408	18,703	
		4096	11970	48,285	11500	927,559	
	SGI POWER CHALLENGE <sup>a</sup> R8000	1024	62.09		8.134		
		2048	497.5		113.8 <sup>b</sup>		
		3030	1538		217.1		
		4096	3978		544.5		
	DEC ALPHA	1024	85.4		N/A		
		2048	710.7		N/A		
		3030	2229		N/A		
2	SGI ONYX R4400	1024	90.92	3	27.75	13	
		2048	737.2	67	408.2	448	
		3030	2548	11,964	1247	27,330	
		4096	<b>6134</b>	24,862	<b>6530</b>	135,241	
	SGI POWER CHALLENGE R8000	1024	30.06		4.825		
		2048	237.1		59.58 <sup>b</sup>		
		3030	773.3		106.5		
		4096	1898		801.2		
	4	SGI ONYX R4400	1024	46.1	3	13.95	0
			2048	376.3	505	204.6	499
			3030	1343	5,537	1298	20,769
			4096	<b>3261</b>	14,259	<b>4218</b>	67,512
SGI POWER CHALLENGE R8000		1024	15.09		2.622		
		2048	120.2		30.52 <sup>b</sup>		
		3030	391.1		54.26		
		4096	953.7		410.2		
8		SGI POWER CHALLENGE R8000	1024	8.33		1.712	
			2048	61.02		15.85 <sup>b</sup>	
			3030	206.3		29.24	
			4096	479.3		216.3	

Table 2: Running times of BLAS3 and QUADTREE

<sup>a</sup>Compare Table 1.<sup>b</sup>See Section 5.3.

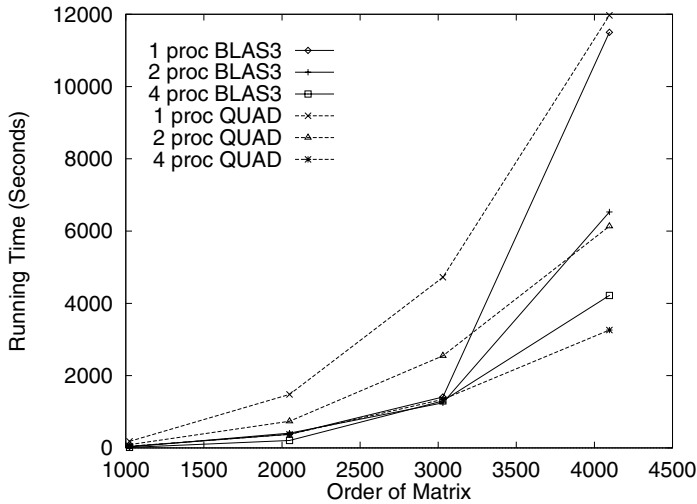


Figure 10: Running times on SGI R4400 ONYX

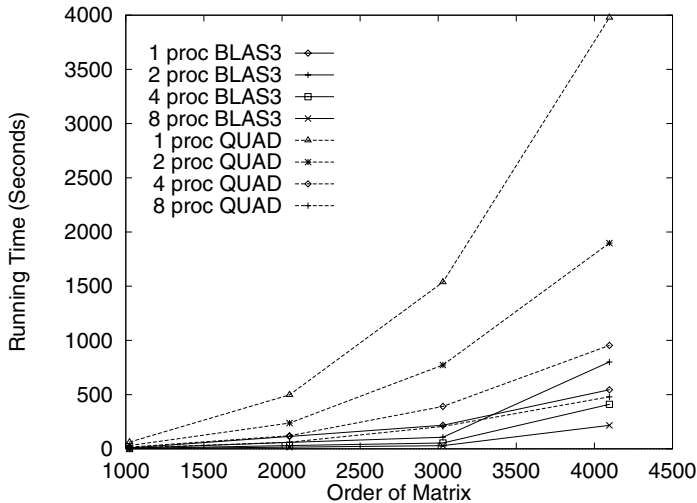


Figure 11: Running times on SGI R8000 POWER CHALLENGE

Several observations arise:

- For matrices of order 4096, BLAS3 was severely crippled by page faults on the ONYX. The QUADTREE algorithm, as designed, made the most of both cache and page reuse, and, thus, it was able to beat the manufacturer's hand-coded BLAS3 routines on two and four processors. This did not occur on the POWER CHALLENGE, whose main memory is large enough to avoid paging.
- The counts of major faults are all from one processor only, even though more were used. Even then, we do not understand why there were so few faults when more processors are active. Certainly it is because of these faults that multiprocessor times of QUADTREE were less than those of BLAS3.
- On the SGI POWER CHALLENGE the QUADTREE algorithm never beats BLAS3 because paging was unnecessary. It closed to within a factor of two on matrices of order 4096 with eight processors. We believe it would win on larger problems.
- In all instances the QUADTREE algorithm exhibits the predicted eight-fold slowdown—or even less—on a problem of double the order. BLAS3 slows much more on larger problems.
- The QUADTREE algorithm on the DEC ALPHA is presented here for relative comparison; BLAS3 routines were not available for comparison. It also exhibits the eight-fold slowdown when the order doubles.

### 5.3 A strange case.

All of the running times for the BLAS3 algorithm on matrices of order 2048 are reproducibly out of line in Table 2 on the SGI POWER CHALLENGE. Under multiprocessing those for order 4096 are also surprisingly slow. This might be due to an exception to its striding strategy, of the sort that we never saw with the QUADTREE algorithm. On a matrix of order 2047, however, the uniprocessing BLAS3 code yields a running time of 69 rather than 114 seconds (Table 1); this value scales with those of the other orders and might be used instead. The anomaly is invisible in Figure 11.

## 6 Analysis

This analysis explains—as supported by experiments over the years [19, 12]—that matrices should be partitioned into square blocks, independently of the size and shape of the problem, for swapping across boundaries in a layered memory.

**Definition 4** A cache miss is a demand for transfer of a block of data, contiguous in memory, to and from similarly contiguous memory elsewhere.

Figure 12 sketches the algorithm for the cumulative block product  $C+ = A \cdot B$ , where  $C$  is  $n \times m$ ,  $A$  is sized  $n \times l$ , and  $B$  is  $l \times m$ . We seek optimal values for the blocking within each of the matrices, respectively  $r \times q$ ,  $r \times p$ , and  $p \times q$ , to minimize the number of misses across the algorithm.

**Constraint 1**  $l, m, n \gg p, q, r$ .

**Definition 5** Let  $s$  be the capacity of data cache, measured in units of (floating-point) Scalars.

**Constraint 2**  $s, p, q, r > 0$ .

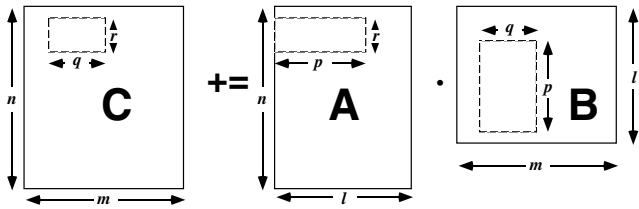


Figure 12:  $C = A * B$  with  $C$ 's blocks sized  $r \times q$ ;  $A$ 's blocks  $r \times p$ ; and  $B$ 's,  $p \times q$ .

**Constraint 3**  $pq + qr + rq \leq s$ .

**Definition 6** The function  $f$  maps  $\langle l, p, m, q, n, r \rangle$  (labeled according to Figure 12) to the number of scalars transferred to cache in a block multiply as illustrated in Figure 12.

The image of  $f$  is proportional to the traditional FLOP-count, usually  $lmn$ . For the most common iterative algorithms of three nested `for`-loops (cf. Figure 1), it is well known that only two cache misses are necessary between the innermost steps. Optimal block sizes seem to depend on all of  $l, m, n, p, q, r$ ; we seek good values for the last three, independent of the first three.

The six familiar algorithms [14, p. 19] for the product (two using inner-, two middle-, and two outer-products), correspond to each of the six permutations of the three `for`-loops in Figure 1. The first theorem abstracts the inner-product version where the  $\langle l, p \rangle$  dimension is traversed in the inner of three loops, and  $\langle n, r \rangle$  is traversed in the outer. Abstracting any of the other five would arrive at the same formula with the pairs  $\{\langle l, p \rangle, \langle m, q \rangle, \langle n, r \rangle\}$  permuted according to the rearrangement of the loops.

**Theorem 1**

$$f(l, p, m, q, n, r) \gtrsim \frac{q+r}{qr} lmn + \frac{q-p}{q} mn + \frac{p}{r}(r-q)n + pq.$$

**Proof:** Two of the three blocks for an inner-product are swapped at a time; read-only  $r \times p$  blocks from  $A$  and  $p \times q$  blocks  $B$  are loaded in the inner `for`-loop. At the end of that loop, however, the two blocks are taken from  $C$  and  $B$  (but not from  $A$  if the code is clever enough to reverse the direction of alternate inner loops), and at the end of the middle loop from  $C$  and  $A$  (similarly). The last term, below, accounts for the initial loading of cache.

$$\begin{aligned} f(l, p, m, q, n, r) &= \left\lceil \frac{n}{r} \right\rceil \left\lceil \frac{m}{q} \right\rceil \left( \left\lceil \frac{l}{p} \right\rceil - 1 \right) (rp + pq) \\ &+ \left\lceil \frac{n}{r} \right\rceil \left( \left\lceil \frac{m}{q} \right\rceil - 1 \right) (rq + pq) \\ &+ \left( \left\lceil \frac{n}{r} \right\rceil - 1 \right) (rq + rp) + (rq + rp + pq). \end{aligned}$$

Using Constraint 1, we drop the ceiling functions:

$$\begin{aligned} f(l, p, m, q, n, r) &\gtrsim \frac{n}{r} \frac{m}{q} \left( \frac{l}{p} - 1 \right) (rp + pq) \\ &+ \frac{n}{r} \left( \frac{m}{q} - 1 \right) (rq + pq) \\ &+ \left( \frac{n}{r} - 1 \right) (rq + rp) + (qr + pr + pq). \blacksquare \end{aligned}$$

This result does not yield a broadly useful minimum for  $p, q, r$  because it depends on the relative sizes  $l, m, n$  of the matrices.

**Definition 7** The function  $g$  maps  $\langle l, p, m, q, n, r \rangle$  to the number of cache misses (processor with one duplex memory port) in the block multiply illustrated in Figure 12.

**Theorem 2**  $g(l, p, m, q, n, r) \gtrsim 2 \frac{lmn}{pqr} + 1$ .

**Theorem 3** The minimal value of  $g(l, p, m, q, n, r)$ , independent of  $\langle l, m, n \rangle$ , occurs near  $p = q = r = \sqrt{s/3}$ .

**Proof:** From Theorem 2, we need to maximize  $pqr$  such that  $s \geq p, q, r > 0$  and  $pq + qr + rp \leq s$ . This problem can be interpreted as maximizing the volume of a  $p \times q \times r$  rectangular solid, subject to the constraint that its surface area is less than  $2s$ . The maximum occurs when all of  $p, q, r$  are  $\sqrt{s/3}$ : a cube. Of course, the integral constraints and ceiling functions temper this solution relative to  $l, m, n$ . So, optimal blocks are nearly—not always perfectly (using a Lagrange multiplier)—square, independently of  $l, m$ , and  $n$ .  $\blacksquare$

This establishes the apocryphal result: square (or nearly square) blocks offer optimal locality, independently of whether  $A, B$ , or  $C$  are rectangular.

## 7 Conclusions

The cache-miss metric is artificial. It does not explain the magnitudes of most run times that we produced. On the other hand, it does predict their relative ordering, and also that decomposition into square subproblems reduces swapping, and that communication time should be considered *first* on modern architectures. That prediction, alone, is responsible for the dovetailed code in Figure 7; if the only impact of an abstract analysis were to inspire a good algorithm, then this one has already established itself.

On one processor our quadtree algorithm runs about 3.5 times faster than inner-product codes and only 7 or 8 times slower than uniprocessor BLAS3. Under multiprocessing, however, BLAS3 slows and—with all its disadvantages—the recursive quadtree algorithm was seen to overtake it. We attribute the reversal to the time required for swapping to/from secondary cache, which was not foreseen in tuning the BLAS3 code. On the ONYX the swap was paging between disk and RAM.

Some of our experiments were run on computers in a quiet environment—without competing processes; this was important on the ONYX. Earlier tests suggested corruption, even of large caches, from competing programs. Readers are cautioned to seek solitude for accurate cache-watching.

Similarly, we caution programmers that recursion is excessively awkward on compilers for super-scalar processors that do not unfold the base cases in the same way that they do unroll inner loops. These two insights—code dovetailed for cache reuse and unfolding of its base cases for superscalar architectures—alone account for almost a four-fold improvement in running times during the course of our development.

We have not tested the attractive hybrid composed of Strassen's recurrence and this one; stability [17] and in-place constraints that are already satisfied here would have to be relaxed there. However, for the balanced, dense matrix (Case 0) Strassen's original 18-addition presentation also bifurcates nicely into 3-and-3 parallel block multiplications, followed by 1 serially. That suggests a hybrid scheme in which our indexing and recursion is used at the highest, unbalanced levels (Cases 1–7) and Strassen's is used on the intermediate, balanced subproblems. Ours again becomes appropriate for the tiny blocks [20] that fit in cache. Figure 4's indexing, of course, is used throughout any recurrence—of these two or of other algorithms.

The underlying matrix representation is critical to these results. Indexing the matrix according to the level-order traversal of its quadtree decomposition has the effect of localizing every subtree/block in contiguous addresses. The effect is acceleration of the computation by reuse of whichever block happens to fit into a cache or a page—without anyone specifying the size of that block. The code is efficient and portable because it takes good advantage of that locality.

The use of this structure and this indexing should not be limited to this particular problem. Other solutions can take good advantage of it. For instance, in *LU* decomposition it provides convenient pivoting on any block that is a subtree [24], whose elimination needs only a single traversal using exactly this multiplication. We are finding that other classic algorithms can also take good advantage of the structure, the indexing, and the recursive style.

## 8 Addendum

Since this paper was completed, Version 7.1 of the SGI C compiler was installed on our site. It has the effect of *increasing* the times for the quadtree algorithm over those reported here.

Also since then, we experimented with the quadrant ordering within the representation, without changing other code. It is presented above as northwest, southwest, southeast, northeast (called the “U” ordering.) For reasons not yet understood, an older ordering: northwest, northeast, southwest, southeast (called the “Z” ordering) seems to provide better times. Simply changing the data structure from the U ordering to the Z ordering yields a 4–6% improvement over the times published here. And that improvement was obtained under the Version 7.1 compiler.

## 9 Acknowledgements

We thank Randy Bramley and Paul Purdom for comments.

## References

- [1] J. M. Anderson, S. P. Amaralinghe, & M. S. Lam. Data and computation transformations for multiprocessors. *Proc. 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, SIGPLAN Notices* **30**, 8 (August 1995), 166–178.
- [2] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, & M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. *Proc. 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, SIGPLAN Notices* **30**, 8 (August 1995), 1–17.
- [3] F. W. Burton & J. G. Kollias. Comment on ‘The explicit quad tree as a structure for computer graphics.’ *Comput. J.* **26**, 2 (May 1983), 188.
- [4] D. Cann. Retire FORTRAN? : a debate rekindled. *Comm. ACM* **35**, 8 (August 1992), 81–89.
- [5] S. Carr & R. B. Lehoucq. Compiler Blockability of dense matrix factorizations. *ACM Trans. Math. Software* (to appear).
- [6] M. Cierniak & W. Li. Unifying data and control transformations for distributed shared-memory machines. *Proc. ACM SIGPLAN ’95 Conf. on Programming Lang. Design and Implementation, SIGPLAN Notices* **30**, 6 (June 1995), 205–217.
- [7] J. Cohen & M. Roth. On the implementation of Strassen’s fast multiplication algorithm. *Acta Informat.* **6**, 4 (August 1976), 341–355.
- [8] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, & T. von Eicken. LogP: a practical model of parallel computation. *Comm. ACM* **39**, 11 (November 1996), 78–85.
- [9] J. W. Demmel & N. J. Higham. Stability of block algorithms with fast Level 3 BLAS. *ACM Trans. Math. Software* **18**, 3 (September 1992), 274–291.
- [10] J. Dongarra, J. DuCroz, S. Hammarling, & R. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.* **14** (1988), 1–17.
- [11] R. J. Fateman. Symbolic mathematics system evaluators (extended abstract). In Y. N. Lakshman (ed.), *Proc 1996 Intl. Symp. on Symbolic and Algebraic Computation*, New York, ACM Press, 86–94.
- [12] P. C. Fischer & R. L. Probert. Storage reorganization techniques for matrix computation in a paging environment. *Comm. ACM* **22**, 7 (July 1979), 405–415.
- [13] J. Frens & D. S. Wise. Matrix inversion using quadtrees implemented in GOFER. Technical Report 433, Computer Science Dept., Indiana University (May 1995).
- [14] K. A. Gallivan, R. J. Plemmons, & A. H. Sameh. Parallel Algorithms for dense linear algebra. *SIAM Review* **32**, 1 (March 1990), 54–135. Reprinted in Gallivan *et al. Parallel Algorithms for Matrix Computation*, Philadelphia, SIAM (1990), 1–82.
- [15] G. H. Golub & C. F. Van Loan. *Matrix Computations* 2nd edition. The Johns Hopkins University Press, Baltimore (1989).
- [16] J. Fasel, P. Hudak, S. Peyton Jones, & P. Wadler (eds.) HASKELL special issue. *ACM SIGPLAN Notices* **27**, 5 (May 1992).
- [17] N. J. Higham. Exploiting fast matrix multiplication within the Level 3 BLAS. *ACM Trans. Math. Software* **16**, 4 (December 1990), 352–368.
- [18] D. E. Knuth. *The Art of Computer Programming I, Fundamental Algorithms* (2nd ed.), Reading, MA, Addison-Wesley, (1973).
- [19] A. C. McKellar & E. G. Coffman, Jr. Organizing matrices and matrix operations for paged-memory systems *Comm. ACM* **12**, 3 (March 1969), 153–165.
- [20] J. Spiess. Untersuchungen des Zeitgewinns durch neue Algorithmen zur Matrix-Multiplikation. *Computing* **17**, 1 (1976), 23–36.
- [21] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.* **13** (1969), 354–356.
- [22] G. L. Steele, Jr. Debunking the “expensive procedure call” myth, or Procedure call implementations considered harmful, or LAMBDA: the ultimate GOTO. *ACM77: Proc. 1977 Annual Conf.*, New York, ACM (1977), 153–162.
- [23] D. S. Wise. Representing matrices as quadtrees for parallel processors (extended abstract). *ACM SIGSAM Bulletin* **18**, 3 (August 1984), 24–25.

[24] D. S. Wise. Undulant block elimination and integer-preserving matrix inversion. *Sci. Comput. Programming* (to appear). Technical Report 418, Computer Science Department, Indiana University (revised, August 1995).

**Appendix:** Slides newly presented at PPoPP '97

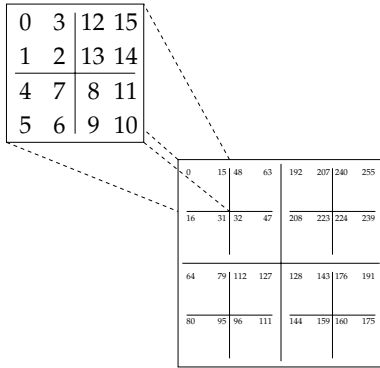


Figure 13: Implementation of quadtree matrices.

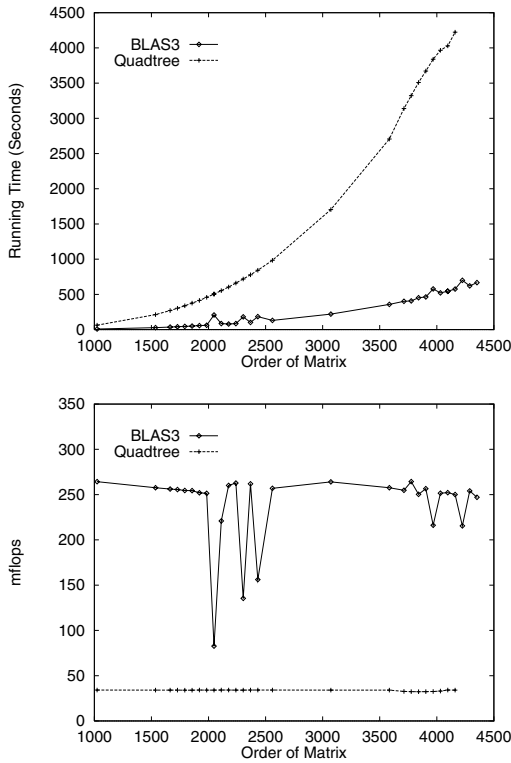


Figure 14: Uniprocessor results on SGI POWER CHALLENGE

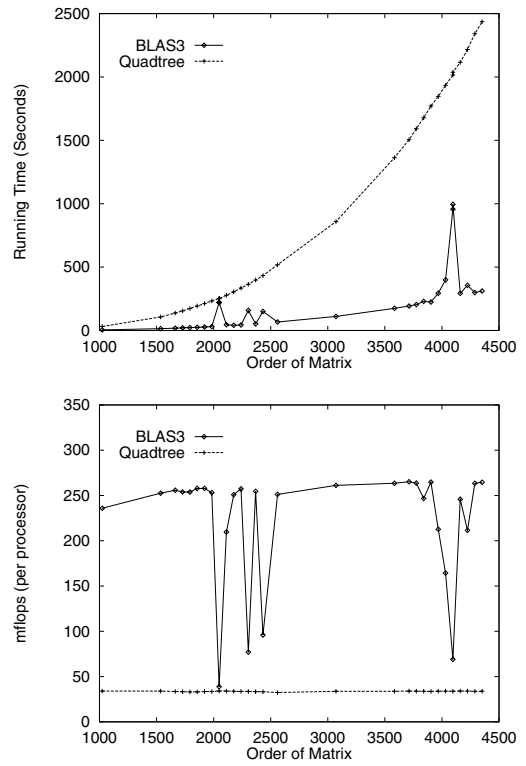


Figure 15: Two-processor results on SGI POWER CHALLENGE

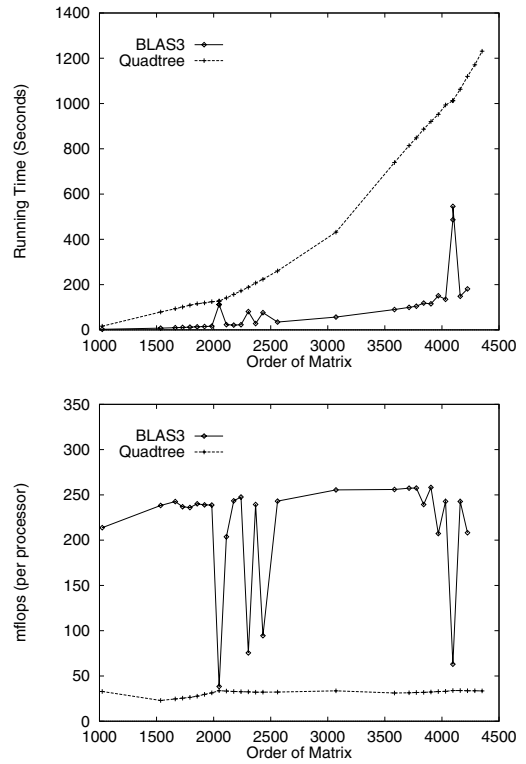


Figure 16: Four-processor results on SGI POWER CHALLENGE

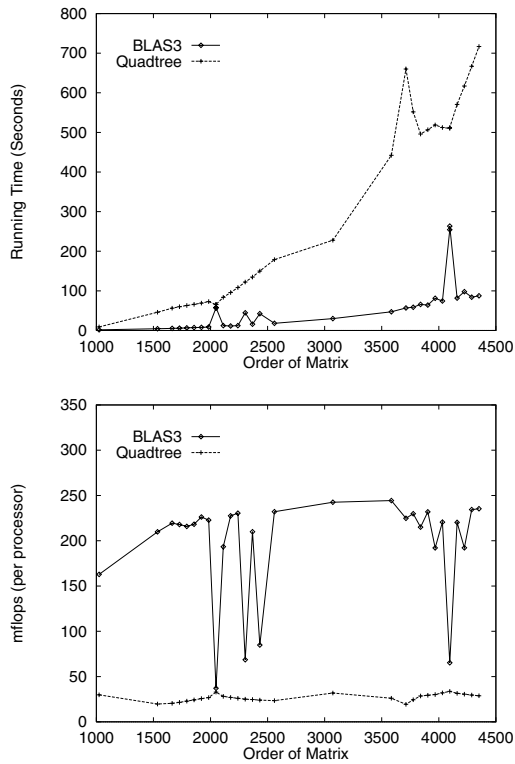


Figure 17: Eight-processor results on SGI POWER CHALLENGE

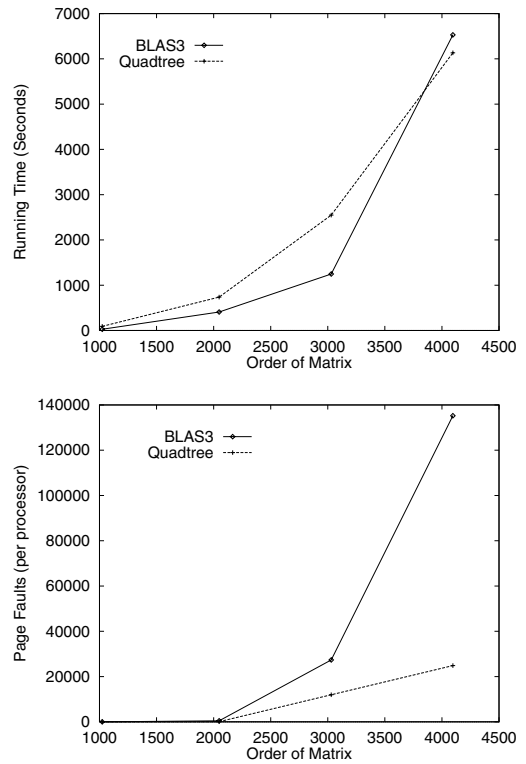


Figure 19: Two-processor results on SGI ONYX

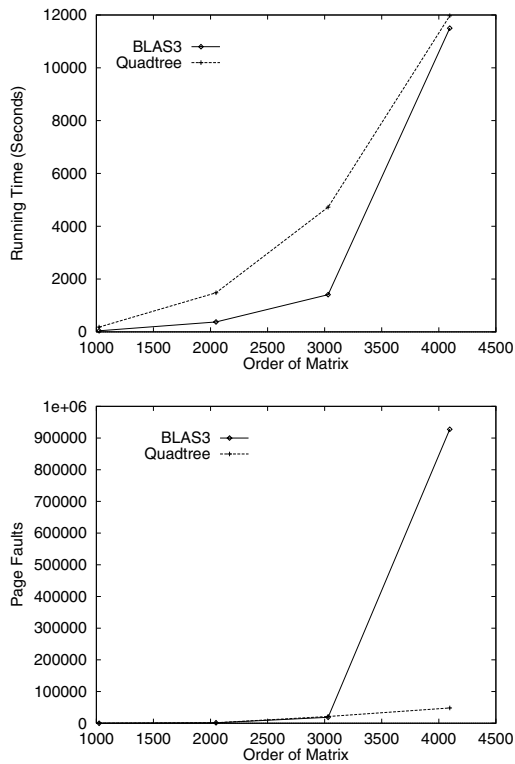


Figure 18: Uniprocessor results on SGI ONYX

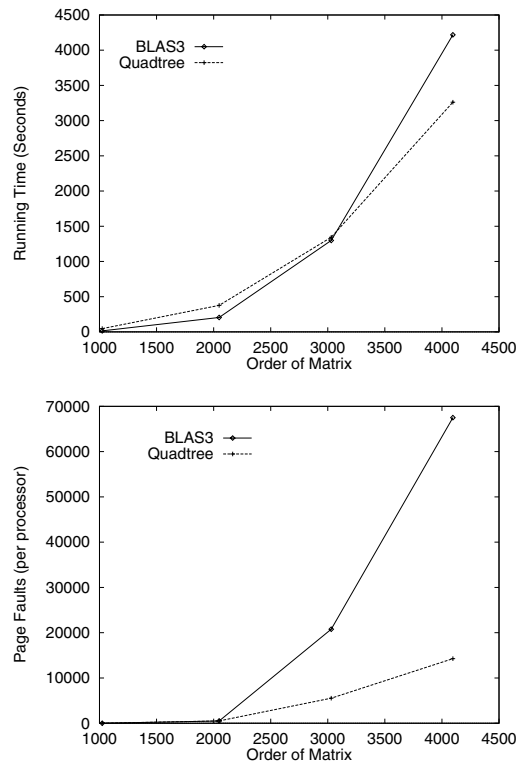


Figure 20: Four-processor results on SGI ONYX