Recursive Array Layouts and Fast Parallel Matrix Multiplication*

Siddhartha Chatterjee† Alvin R. Lebeck‡ Praveen K. Patnala† Mithuna Thottethodi‡

Abstract

Matrix multiplication is an important kernel in linear algebra algorithms, and the performance of both serial and parallel implementations is highly dependent on the memory system behavior. Unfortunately, due to false sharing and cache conflicts, traditional column-major or row-major array layouts incur high variability in memory system performance as matrix size varies. This paper investigates the use of recursive array layouts for improving the performance of parallel recursive matrix multiplication algorithms.

We extend previous work by Frens and Wise on recursive matrix multiplication to examine several recursive array layouts and three recursive algorithms: standard matrix multiplication, and the more complex algorithms of Strassen and Winograd. We show that while recursive array layouts significantly outperform traditional layouts (reducing execution times by a factor of 1.2–2.5) for the standard algorithm, they offer little improvement for Strassen's and Winograd's algorithms; we provide an algorithmic explanation of this phenomenon. We demonstrate that carrying the recursive layout down to the level of individual matrix elements is counterproductive, and that a combination of recursive layouts down to canonically ordered matrix tiles instead yields higher performance. We evaluate five recursive layouts with successively increasing complexity of address computation, and show that addressing overheads can be kept in control even for the most computationally demanding of these layouts. Finally, we provide a critique of the Cilk system that we used to parallelize our code.

1 Introduction

High-performance dense linear algebra codes, whether sequential or parallel, rely on good spatial and temporal locality of reference for their performance. Matrix multiplication is an important kernel in linear algebraic algorithms, and is enshrined in the `dgemm` routine in the BLAS 3 library [10]. There is an intimate relationship between the layout of the arrays in memory and the performance of the routine. On modern shared-memory multiprocessors with multi-level memory hierarchies, the column-major layout assumed in the BLAS 3 library can result in performance anomalies as the matrix size is varied. These anomalies result from unfavorable access patterns in the memory hierarchy that cause interference misses and false sharing and increase memory system overheads experienced by the code. In this paper, we investigate recursive array layouts accompanied by recursive control structures as a means of delivering high and robust performance for parallel dense linear algebra.

The use of quad- or oct-trees (or, in a dual interpretation, space-filling curves) is known in parallel computing [1, 20, 21, 33, 35, 38] for improving both load balance and locality. The computations thus parallelized or restructured are reasonably coarse-grained, thus making the overheads of maintaining and accessing the data structures insignificant. Frens and Wise [12] champion the use of quad-trees to represent matrices and explore its use in matrix multiplication. While the authors presented some excellent ideas, we felt that they carried them to an extreme. Based on this previous work, we were left with several questions that we address in this paper.

- Previous work using such layouts did not worry greatly about the overhead of address computations. The algorithms described in the literature [31] follow from the basic definitions and are not particularly optimized for performance. Are there fast algorithms, perhaps involving bit manipulation, for maintaining the "dope vectors" that would enable such data structures to be used for fine-grained parallelism? Or, even better, can the address computations be embedded implicitly in the control structure of the program?

- Frens and Wise carried out their quad-tree layout of matrices down to the level of matrix elements, disregarding the result of Lam, Rothberg, and Wolf [26] that a tile that fits in cache and is contiguous in memory space can be organized in a canonical order without compromising locality of reference. This suggests the need for the quadtree decomposition to terminate well before the element level and to co-exist with tiles organized in a canonical manner. Could this interfacing of two layout functions be accomplished without increasing the cost of addressing? How does absolute performance relate to the choice of tile size?

- Frens and Wise assumed that all matrices would be organized in quad-tree fashion, and therefore did not quantify the cost of converting to and from a canonical order at the routine interface. This is an optimistic assumption that is currently unrealistic. We feel that an honest accounting of costs needs to quantify the overhead of format conversion. Would the performance benefits of quad-tree data structures be lost in the cost of building them in the first place?

- There are several variants of recursive orderings other than the U-ordering that Frens and Wise used. Some of these

*This work supported in part by DARPA Grant DABT63-98-1-0001, NSF Grants CDA-97-2637 and CDA-95-12356, NSF Career Award MIP-97-02547, The University of North Carolina at Chapel Hill, Duke University, and an equipment donation through Intel Corporation's Technology for Education 2000 Program. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

†Department of Computer Science, The University of North Carolina, Chapel Hill, NC 27599-3175. Email: {sc,praveen}@cs.unc.edu

‡Department of Computer Science, Duke University, Durham, NC 27708-0129. Email: {alvy,mithuna}@cs.duke.edu

orderings, such as Gray-Morton [28] and Hilbert [18], are supposedly better for load balancing, albeit at the expense of greater addressing overhead. How would these variants compare in terms of complexity *vs.* performance improvement for fine-grained parallel computations?

- Frens and Wise speculated about the "attractive hybrid composed of Strassen's recurrence and this one" [12, p. 215]. This is an interesting variation on the problem for two reasons. First, Strassen's algorithm [36] achieves a lower operation count at the expense of more data accesses in patterns that have worse algorithmic locality of reference. Second, the control structure of Strassen's algorithm is not as simple as that of the standard recursive algorithm, making it trickier to work with quad-tree layouts. Could this combination be made to work, and how would it perform?

Our major contributions are as follows. First, we provide improved performance over that reported by Frens and Wise by stopping their recursive splitting well before the level of single elements. Second, we integrate recursive data layouts into Strassen's algorithm and provide some surprising performance results. Third, we test five different recursive layouts and characterize their relative performance. We provide efficient addressing routines for these layout functions that would be useful to implementors wishing to incorporate such layout functions into fine-grained parallel computations. Finally, as a side effect, we provide an evaluation of the strengths and weaknesses of the Cilk system [4], which we used to parallelize our code.

The remainder of this paper is organized as follows. Section 2 introduces the algorithms for fast matrix multiplication that we discuss in this paper. Section 3 introduces recursive data layouts for multi-dimensional arrays. Section 4 describes the implementation issues that arose in weaving together the recursive data layouts with the recursive control structures of the algorithms. Section 5 offers measurement results that support the claim that these layouts improve the overall performance. Section 6 compares our approach with previous related work. Section 7 presents conclusions.

2 Algorithms for fast matrix multiplication

The computation we discuss in this paper is that of matrix multiplication. Let A and B be two $n \times n$ matrices, where $n = 2^k$. Let $C = A \bullet B$, where the symbol $\bullet$ represents matrix multiplication.

Rather than formulate the matrix product in terms of row or column operations, we will proceed by quadrant or sub-matrix operations. Partition the two input matrices *A* and *B* and the result matrix *C* into quadrants as follows.

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \bullet \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad (1)$$

The standard algorithm that performs $O(n^3)$ operations proceeds as shown in Figure 1(a), performing eight recursive matrix multiplication calls and four matrix additions.

Strassen's original algorithm [36] uses algebraic identities to reduce the number of recursive matrix multiplication calls from eight to seven at the cost of 18 matrix additions/subtractions. This change reduces the operation count to $O(n^{\lg 7})$. The algorithm proceeds as shown in Figure 1(b).

Winograd's variant [11] of Strassen's algorithm uses seven recursive matrix multiplication calls and 15 matrix additions/subtractions. It is known [11] that this is the minimum number of multiplications and additions possible for any recursive matrix multiplication algorithm based on division into quadrants. The computation proceeds as shown in Figure 1(c). Compared to Strassen's original algorithm, the noteworthy feature of Winograd's variant is its identification and reuse of common subexpressions. These shared computations are responsible for reducing the number of additions, but also contribute to worse locality of reference unless special attention is given to this aspect of the computation.

Figure 1(a)–(c) also illustrate the elements of A and B read to compute the individual elements of $C = A \bullet B$, for 8×8 matrices. Each of the six diagrams has an 8×8 grid of boxes, each box representing an element of C. Each box contains an 8×8 grid of points, each point representing an element of matrix A or B. The grid points corresponding to elements read are indicated by a dot. From these figures, we observe that the standard algorithm has good *algorithmic locality of reference*, accessing consecutive elements in a matrix row or column.[1] In contrast, the access patterns of Strassen's and Winograd's algorithms are much worse in terms of algorithmic locality. This is particularly evident along the main diagonal for Strassen's algorithm and for elements $(0, 7)$ and $(7, 0)$ for Winograd's. This raises the question of whether the benefits of the reduced number of floating point operations for the fast algorithms would be lost as a result of the increased number of memory accesses. We answer this question in Section 5.

We do not discuss in this paper numerical issues concerning the fast algorithms, as they are covered elsewhere [17].

We use Cilk [4] to implement parallel versions of these algorithms. Parallelism is exposed in the recursive matrix multiplication calls. The seven or eight calls are spawned in parallel, and these in turn invoke other recursive calls in parallel. Cilk supports this nested parallelism, providing a very clean implementation.

2.1 Interface

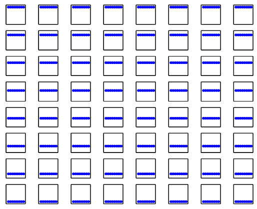
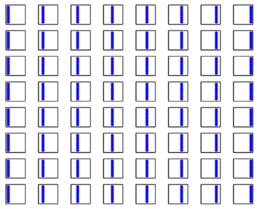
In order to stay consistent with previous work and to permit meaningful comparisons, all our implementations follow the same calling conventions as the `dgemm` subroutine in the Level 3 BLAS library [10]. Thus, the implementation computes $C \leftarrow \alpha * \mathrm{op}(A) \bullet \mathrm{op}(B) + \beta * C$, where α and β are scalars, $\mathrm{op}(A)$ is an $m \times k$ double precision real matrix, $\mathrm{op}(B)$ is a $k \times n$ double precision real matrix, C is an $m \times n$ double precision real matrix, and $\mathrm{op}(X)$ is either X or X^T. The matrices A, B, and C are stored in column-major order, with leading dimensions *ldA*, *ldB*, and *ldC* respectively.

3 Recursive array layouts

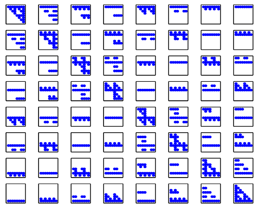
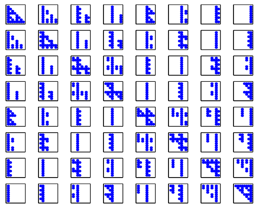
Programming languages that support multi-dimensional arrays must also provide a function (the *layout* function L) to map the array index space into the linear memory address space. For ease of exposition, we assume a two-dimensional array with m rows and n columns, which we index using a zero-based scheme. The results we discuss generalize to higher-dimensional arrays and other indexing schemes. We define L such that $L(i, j)$ is the memory location of the array element in row i and column j relative to the starting memory location of the array, in units of array elements. We list near the end of the argument list of L, following a semicolon, any "structural" parameters (such as m and n) of L, thus: $L(i, j; m, n)$.

The default layout functions provided in current programming languages are the *row-major* layout L_R as used in Pascal, given by $L_R(i, j; m, n) = n \cdot i + j$, and the *column-major* layout L_C as

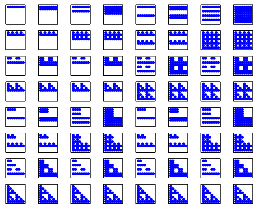
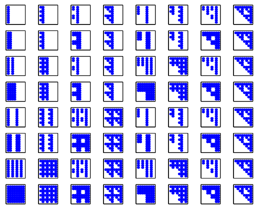
[1] We add the qualifier "algorithmic" to emphasize the point that we are reasoning about this issue at an algorithmic level, independent of the architecture of the underlying memory hierarchy. In terms of the 3C model [19] of cache misses, we are reasoning about capacity misses at a high level, not about conflict misses.

Post-additions	Recursive calls	Pre-additions	Elts of A read to compute C	Elts of B read to compute C
$C_{11} = P_1 + P_2$ $C_{21} = P_3 + P_4$ $C_{12} = P_5 + P_6$ $C_{22} = P_7 + P_8$	$P_1 = A_{11} \bullet B_{11}$ $P_2 = A_{12} \bullet B_{21}$ $P_3 = A_{21} \bullet B_{11}$ $P_4 = A_{22} \bullet B_{21}$ $P_5 = A_{11} \bullet B_{12}$ $P_6 = A_{12} \bullet B_{22}$ $P_7 = A_{21} \bullet B_{12}$ $P_8 = A_{22} \bullet B_{22}$			

(a) Standard algorithm

Post-additions	Recursive calls	Pre-additions	Elts of A read to compute C	Elts of B read to compute C
$C_{11} = P_1 + P_4 - P_5 + P_7$ $C_{21} = P_2 + P_4$ $C_{12} = P_3 + P_5$ $C_{22} = P_1 + P_3 - P_2 + P_6$	$P_1 = S_1 \bullet T_1$ $P_2 = S_2 \bullet B_{11}$ $P_3 = A_{11} \bullet T_2$ $P_4 = A_{22} \bullet T_3$ $P_5 = S_3 \bullet B_{22}$ $P_6 = S_4 \bullet T_4$ $P_7 = S_5 \bullet T_5$	$S_1 = A_{11} + A_{22}$ $S_2 = A_{21} + A_{22}$ $S_3 = A_{11} - A_{12}$ $S_4 = A_{21} - A_{11}$ $S_5 = A_{12} - A_{22}$ $T_1 = B_{11} + B_{22}$ $T_2 = B_{12} - B_{22}$ $T_3 = B_{21} - B_{11}$ $T_4 = B_{11} + B_{12}$ $T_5 = B_{21} + B_{22}$		

(b) Strassen's algorithm

Post-additions	Recursive calls	Pre-additions	Elts of A read to compute C	Elts of B read to compute C
$C_{11} = \; U_1 = P_1 + P_2$ $U_2 = P_1 + P_4$ $U_3 = U_2 + P_5$ $C_{21} = \; U_4 = U_3 + P_7$ $C_{22} = \; U_5 = U_3 + P_3$ $U_6 = U_2 + P_3$ $C_{12} = \; U_7 = U_6 + P_6$	$P_1 = A_{11} \bullet B_{11}$ $P_2 = A_{12} \bullet B_{21}$ $P_3 = S_1 \bullet T_1$ $P_4 = S_2 \bullet T_2$ $P_5 = S_3 \bullet T_3$ $P_6 = S_4 \bullet B_{22}$ $P_7 = A_{22} \bullet T_4$	$S_1 = A_{21} + A_{22}$ $S_2 = S_1 - A_{11}$ $S_3 = A_{11} - A_{21}$ $S_4 = A_{12} - S_2$ $T_1 = B_{12} - B_{11}$ $T_2 = B_{22} - T_1$ $T_3 = B_{22} - B_{12}$ $T_4 = B_{21} - T_2$		

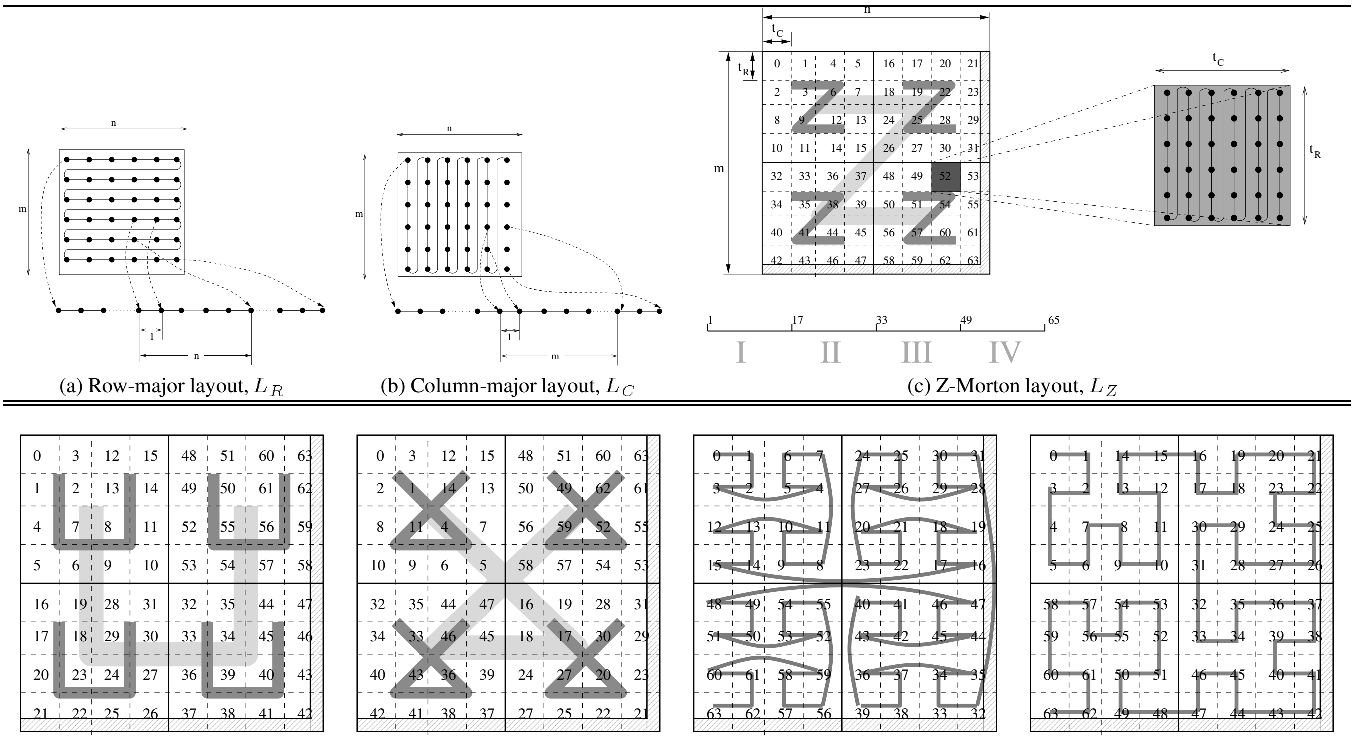
(c) Winograd's variant of Strassen's algorithm

Figure 1: Descriptions and algorithmic locality of reference of the three matrix multiplication algorithms.

used in Fortran, given by $L_C(i, j; m, n) = m \cdot j + i$. We do not consider the *vector-of-vector* layout used for non-constant arrays in C, because these are not true multi-dimensional arrays. Following the terminology of Cierniak and Li [8], we refer to L_R and L_C as *canonical* layout functions. Figure 2(a) and (b) show these two layout functions. Canonical layouts do not always interact well with cache memories, because the layout function favors one axis of the index space over the other: neighbors in the unfavored direction become distant in memory. This *dilation* effect has implications for both parallel execution and single-node performance that can reduce program performance. In the shared-memory parallel environments in which we experimented, the elements of a quadrant of a matrix are spread out in shared memory, and a single shared memory block can contain elements from two quadrants, and thus be written by the two processors computing those quadrants. This leads to false sharing [9]. In a message-passing parallel environment such as those used in implementations of High Performance Fortran [25], typical array distributions would again spread a matrix quadrant over many processors, thereby increasing communication costs. The dilation effect can compromise single-node memory system performance in the following ways: by reducing or even nullifying the effectiveness of multi-word cache lines; by reducing the effectiveness of translation lookaside buffers (TLBs) for large matrix sizes; and by causing cache misses due to self-interference even when a tiled loop repeatedly accesses a small array tile.

The above considerations have led authors to investigate a class of array layout functions variously described as being based either on quadtrees [12] or on space-filling curves [18, 32, 34]. Instances of this family are familiar in parallel computing under the names *Morton ordering* and *Hilbert ordering*.

Despite the dilation effect described above, canonical layout functions have one major advantage: they allow quick incremental address computation. This idiom is understood by compilers and is one of the keys to high performance in libraries such as native BLAS. Unlike Frens and Wise, therefore, we will not carry the recursive layout down to the level of individual elements, but



(a) Row-major layout, L_R (b) Column-major layout, L_C (c) Z-Morton layout, L_Z

(d) U-Morton layout, L_U (e) X-Morton layout, L_X (f) G-Morton layout, L_G (g) Hilbert layout, L_H

Figure 2: Graphical description of layout functions. Arrays are $m \times n$; tiles are $t_R \times t_C$.

will terminate the recursion when we reach a $t_R \times t_C$ submatrix that fits in cache. We will also describe these layouts in terms of space-filling curves rather than in terms of quad-trees. (The quad-tree is simply an aid to conceptualizing the layout, and none of its internal nodes need to be physically represented in memory. The space-filling curve makes this clear.)

Assume for the moment that we know how to choose t_R and t_C, and that t_R and t_C simultaneously satisfy

$$\frac{m}{t_R} = \frac{n}{t_C} = 2^d \tag{2}$$

for some positive integer d. We can now view our original $m \times n$ array as a $\frac{m}{t_R} \times \frac{n}{t_C}$ array of $t_R \times t_C$ tiles. Equivalently, we have mapped the original two-dimensional array index space (i, j) into the four-dimensional space

$$\begin{aligned}
(t_i, t_j, f_i, f_j) &= (\mathbb{T}(i;t_R), \mathbb{T}(j;t_C), \mathbb{F}(i;t_R), \mathbb{F}(j;t_C)) \\
\mathbb{T}(i;t) &= i \operatorname{div} t \\
\mathbb{F}(i;t) &= i \bmod t.
\end{aligned}$$

We now create two subspaces: the space T of tile co-ordinates (t_i, t_j), and the space F of tile offsets (f_i, f_j). We apply the canonical layout function L_C in the F-space (to keep each tile contiguous in memory) and a layout function L_T in the T-space (to obtain the starting memory location of the tile), and define our recursive layout function L as their sum:

$$\begin{aligned}
L(i, j; m, n, t_R, t_C) &= L(t_i, t_j, f_i, f_j; m, n, t_R, t_C) \\
&= L_T(t_i, t_j; m, n, t_R, t_C) \\
&\quad + L_C(f_i, f_j; t_R, t_C) \\
&= t_R t_C \mathbb{S}(t_i, t_j) + L_C(f_i, f_j; t_R, t_C) \quad (3)
\end{aligned}$$

where $\mathbb{S}(i, j)$ gives the position along the space-filling curve (*i.e.*, the pre-image) of the element at rectangular co-ordinates (i, j).

Equation (3) defines a family of layout functions parameterized by the function $\mathbb{S}$ characterizing the space-filling curve. All of these recursive layout functions have the following operational interpretation following from Hilbert's construction [18]. Divide the original matrix into four quadrants, and lay out these submatrices in memory in an order specified by L_T. A $k_R \times k_C$ submatrix with $k_R > t_R$ and $k_C > t_C$ is laid out recursively using L_T; a $t_R \times t_C$ tile is laid out using L_C.

Space-filling curves are based on the idea of threading a region with self-similar line segments at multiple scales. The two fundamental operations involved are scaling and orienting (rotating) the line segments. We classify the five recursive layouts we consider in this paper into three classes based on the number of orientations needed. Three layouts (U-Morton, X-Morton, and Z-Morton) require a single orientation; one layout (Gray-Morton) requires two orientations; and one layout (Hilbert) requires four orientations. We now discuss the structure of these layouts and the computations involved in calculating their $\mathbb{S}$ functions.

We need the following notation to discuss the computational aspects of the recursive layouts. For any non-negative integer i, let $\mathcal{B}(i)$ be the bit string corresponding to its standard binary encoding, and let $\mathcal{G}(i)$ be the bit string corresponding to its Gray code [30] encoding. Correspondingly, for any bit string s, let $\mathcal{B}^{-1}(s)$ be the non-negative integer i such that $\mathcal{B}(i) = s$, and let $\mathcal{G}^{-1}(s)$ be the non-negative integer i such that $\mathcal{G}(i) = s$, Also, if $u = u_{d-1} \cdots u_0$ and $v = v_{d-1} \cdots v_0$ are two bit patterns each of length d, let $u \bowtie v$ be the bit pattern of length $2d$ resulting from the bitwise interleaving of u and v, *i.e.*, $u \bowtie v = u_{d-1}v_{d-1} \cdots u_0v_0$. Finally, we adopt the convention that, for all layouts, $\mathbb{S}(0, 0) = 0$. Rotations and reflections of the layout functions are possible, and are most cleanly computed by interchanging the i and j arguments and/or subtracting them from $2^d - 1$.

3.1 Recursive layouts with a single orientation

The three layouts U-Morton (L_U), X-Morton (L_X), and Z-Morton (L_Z), illustrated in Figure 2(c)–(e), are based on a single pattern of ordering quadrants that repeat *ad infinitum*. The mnemonics are based on the letters of the English alphabet that these ordering patterns resemble. We note that the Z-Morton layout should properly be called the Lebesgue layout, since it is based on Lebesgue's space-filling curve [34, p. 80].

The $\mathbb{S}$ functions for these layouts are easily computed with bit operations, as follows.

For L_U: $\mathbb{S}(i,j) = \mathcal{B}^{-1}(\mathcal{B}(j) \bowtie (\mathcal{B}(i)\,\mathrm{XOR}\,\mathcal{B}(j)))$
For L_X: $\mathbb{S}(i,j) = \mathcal{B}^{-1}((\mathcal{B}(i)\,\mathrm{XOR}\,\mathcal{B}(j)) \bowtie \mathcal{B}(j))$
For L_Z: $\mathbb{S}(i,j) = \mathcal{B}^{-1}(\mathcal{B}(i) \bowtie \mathcal{B}(j))$

3.2 Recursive layouts with two orientations

The Gray-Morton layout [28] (L_G) is based on a C-shaped line segment and its counterpart that is rotated by 180 degrees. Figure 2(f) illustrates this layout. Computationally, its $\mathbb{S}$ function is defined as follows: $\mathbb{S}(i,j) = \mathcal{G}^{-1}(\mathcal{G}(i) \bowtie \mathcal{G}(j))$.

3.3 Recursive layouts with four orientations

The Hilbert layout [18] (L_H) is based on a C-shaped line segment and its three counterparts rotated by 90, 180, and 270 degrees. Figure 2(g) illustrates this layout. The $\mathbb{S}$ function for this layout is computationally more complex than any of the others. The fastest method we know of is based on an informal description by Bially [2], which works by driving a finite state machine with pairs of bits from i and j, delivering two bits of $\mathbb{S}(i,j)$ at each step.

3.4 Summary

We have described five recursive layout functions in terms of space-filling curves. These layouts grow in complexity from the ones based on Lebesgue's space-filling curve to the one based on Hilbert's space-filling curve. We now state several facts of interest regarding the mathematical and computational properties of these layout functions.

- It follows from the pigeonhole principle that only two of the four cardinal neighbors of (i,j) can be adjacent to $\mathbb{S}(i,j)$. Thus, recursive layouts are prone to a dilation effect similar to that the canonical layouts experience. The important difference is that the dilation occurs at multiple scales. This dilation effect is seen in the abrupt jumps in the curves of Figure 2. We note that these jumps get less pronounced as the number of orientations increases.

- The L_G layout function has a useful symmetry that is easiest to appreciate visually. Refer to Figure 2(f) and observe the northwest quadrant (tiles 0–15) and the southeast quadrant (tiles 32–47), which have different orientations. If we remove the single edge between the top half and the bottom half of each quadrant (edge 7–8 for the northwest quadrant, edge 39–40 for the southeast quadrant), we note that the top and bottom halves of the two quadrants are identically oriented. That is, two quadrants of opposite orientation differ only in the order in which their top and bottom halves are "glued" together. We exploit this symmetry in Section 4.

- In terms of computational complexity of the $\mathbb{S}$ functions of the different layout functions, we observe that bits $2u+1$ and $2u$ of $\mathbb{S}(i,j)$ depend only on bit u of i and j for the layouts with a single orientation, while they depend on bits i through $d-1$ for the L_G and L_H layouts.

4 Implementation issues

Section 2 described the parallel recursive control structure of the matrix multiplication algorithms, and Section 3 described the recursive array layouts we wanted to combine with the algorithms. This section links these two aspects together, addressing a number of implementation issues in the process.

Integration of address computation into control structure. For the matrix multiplication algorithms of Section 2, we can integrate the computation of the $\mathbb{S}$ function into the control structure in an incremental manner, as follows. Observe that the actual work of matrix multiplication happens on $t_R \times t_C$ tiles when the recursion terminates. At each recursive step, we need to locate the quadrants of the current trio of matrices, perform pre-additions on quadrants, spawn the parallel recursive calls, and perform the post-additions on quadrants. The additions have no temporal locality and are ideally suited to streaming through the memory hierarchy, which is aided by the fact that recursive layouts keep quadrants contiguous in memory. Therefore, all we need is the ability to quickly locate the starting points of the four quadrants of a (sub)matrix. This produces the correct $\mathbb{S}$ number of the tiles when the recursion terminates, which are then converted to memory addresses and passed to the leaf matrix multiplication routine.

For the L_G and L_H layout functions, which have multiple orientations, we need to retain both the location and the orientation of quadrants. We encode orientation in one or two most significant bits of the integers.

Relaxing the constraint of equation (2). The definitions of the recursive matrix layouts in Section 3 assumed that t_R and t_C were constrained as described in equation (2). This assumption does not hold in general, and the conceptual way of fixing this problem is to pad the matrix to an $m' \times n'$ matrix which satisfies equation (2). There are two concrete ways to implement this padding process.

- Frens and Wise keep a flag at internal nodes of their quad-tree representation to indicate empty or nearly full subtrees, which "directs the algebra around zeroes (as additive identities and multiplicative annihilators)" [12, p. 208].

  Maintaining such flags makes their solution insensitive to the amount of padding, but requires maintaining the internal nodes of the quad-tree. This scheme is particularly useful for sparse matrices, where patches of zeros can occur in arbitrary portions of the matrices. Note that if one carries the quad-tree decomposition down to individual elements, then $m' \approx 2m$ and $n' \approx 2n$ in the worst case.

- We pick t_R and t_C from an architecture-dependent range, explicitly insert the zero padding, and blindly perform all the arithmetic on the zeros. We choose the range of acceptable tile sizes so that the tiles are neither too small (which would increase the overhead of recursive control) nor overflow the cache (which would result in capacity misses). Since tiles are contiguous, there are no self-interference misses. This makes the performance of the leaf-level matrix multiplications almost insensitive to the tile size [37].

Our scheme is sensitive to the amount of padding, since it performs computations on the padded portions of the matrices. However, if we choose tile sizes from the range $[T_{\min}, T_{\max}]$, the maximum ratio of pad to matrix size is $1/T_{\min}$.

Our imposition of the range $[T_{\min}, T_{\max}]$ of tile sizes is guided by cache considerations, but causes a problem for rectangular matrices whose aspect ratio m/n is either too large or too small. Let $\alpha = T_{\max}/T_{\min}$, and call a matrix *wide*, *squat*, or *lean* depending on whether $\alpha < m/n$, $1/\alpha \leqslant m/n \leqslant \alpha$, or $1/\alpha > m/n$. For wide and lean matrices, it is not possible to find tile sizes that simultaneously satisfy the constraints of equation (2) and lie in the prescribed range of tile sizes.[2] This problem can easily be understood by considering the following case: $m = 1024$, $n = 256$, $T_{\min} = 17$, and $T_{\max} = 32$.

We overcome this limitation quite simply, by dividing wide or lean the matrix into squat submatrices, and reconstructing the matrix product in terms of the submatrix products. Figure 3(a) and Figure 3(b) show two examples of how the input matrices A and B are divided, and how the result C is reconstructed from results of submatrix multiplications. These multiple submatrix multiplications are, of course, spawned to execute in parallel.

Conversion and transposition issues. In order to stay compatible with `dgemm`, we assume that all matrices are presented in column-major layout. Our implementation internally allocates additional storage and converts the matrices from column-major to the recursive layout. The remapping of the individual tiles is again amenable to parallel execution. We incorporate any matrix transposition operations required by the operation into this remapping step. This is handy, because it requires only a single routine for the core matrix multiplication algorithm. The alternative solution requires multiple code versions or indirection using pointers to handle these cases correctly.

Issues with pre- and post-additions. There is one final implementation issue arising from the interaction of the pre- and post-additions with the recursive layouts with more than one orientation. Consider, for example, the addition $S_1 = A_{11} + A_{22}$ in Strassen's algorithm. For recursive layouts with a single orientation, we simply stream through the appropriate number of elements from the starting locations of A_{11} and A_{22}, adding them and streaming them out to S_1. This is not true for L_G and L_H layouts, since the orientations of A_{11} and A_{22} are different. In other words, while each tile (and the entire set of tiles) is contiguous in memory, corresponding tiles of A_{11} and A_{22} are not at the same relative position.

For L_G, the fix is simple, exploiting the symmetry discussed in Section 3.4. If the orientations of the two tiles are different, and each quadrant contains $2k$ tiles, then the ordering of tiles in one orientation is $T_1, \ldots, T_{2k}$ while the ordering of tiles in the other orientation is $T_{k+1}, \ldots, T_{2k}, T_1, \ldots, T_k$. Therefore, we simply need to perform the pre- and post-additions in two half-steps.

For L_H, the situation is more complicated, because there is no simple pattern to the ordering of tiles. Instead, we keep global mapping arrays of these orders for the various orientations, and use these arrays to identify corresponding tiles in pre- and post-additions. The added cost in loop control is insignificant.

[2] The proof is by contradiction. Let $m/n > \alpha$, and assume that we can find t_R and t_C as desired. From equation (2), we have $m/n = t_R/t_C$. Combining the range constraints and equation (2), we get $T_{\min} \leqslant t_C < t_R \leqslant T_{\max}$. But this gives $t_R/t_C < \alpha$. The case $1/\alpha > m/n$ is analogous.

5 Experimental results

Our experimental platform was a Sun Enterprise 3000 SMP with four 170 MHz UltraSPARC processors, and 384 MB of main memory, running SunOS 5.5.1. We used version 5.2.1 of the Cilk system [4] compiled with critical path tracking turned off. The Cilk system requires the use of `gcc` to compile the C files its `cilk2c` compiler generates from Cilk source. We used the gcc-2.7.2 compiler with optimization level `-O3`. The experimental machine was otherwise idle. We also took multiple measurements of every data point to further reduce measurement uncertainty.

We timed the full cross-product of the three algorithms (standard, Strassen, Winograd) and the six layout functions (L_C, L_U, L_X, L_Z, L_G, L_H) running on 1, 2, and 4 processors on square matrices with n ranging from 500 through 1500. We verified correctness of our codes by comparing their outputs with the output of vendor-supplied native version of `dgemm`, However, we could not perform the leaf-level multiplications in our codes by calling the vendor-supplied native version of `dgemm`, since we could not get Cilk to support such linkage of external libraries. Instead, we coded up a C version of a 6-loop tiled matrix multiplication routine with the innermost accumulation loop unrolled four-way. We report all our results in terms of execution time, rather than megaflop/s (which would not correctly account for the padding we introduce) or speedup (since the values are sensitive to the baseline).

General comments. As predicted by theory, we observed the two fast algorithms consistently outperforming the standard algorithm. This is apparent from the different y-axis extents in the subgraphs of Figure 6. From the same figure, we observe virtually no difference between the execution times of the two fast algorithms. This suggests to us that the worse algorithmic locality of reference of Winograd's algorithm compared to Strassen's (see Figure 1) offsets its advantage of lower operation count.

We observed near-perfect scalability for all the codes, as evident from Figures 5 and 6. By running under a different version of Cilk in which critical path tracing was enabled, we determined that, for $n = 1000$, there is sufficient parallelism in the standard algorithm to keep about 40 processors busy; the corresponding number for the two fast algorithms is around 23. This is as expected, since the total work of the algorithms is $O(n^{2+\delta})$, while the critical path is $O(\lg^2 n)$.

Choice of tile size. To back our claim that, for best performance, the recursive layouts should be stopped before the matrix element level, we timed a version of the standard algorithm with the L_Z layout in which we explicitly controlled the tile size at which the recursive layout stopped. Figure 4 shows the single-processor execution times from this experiment with: $n = 1024$, $t \in \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512\}$; and $n = 1536$, $t \in \{3, 6, 12, 24, 48, 96, 192, 384, 768\}$. (We used these values of n because they allow us to choose many tile sizes without incurring any padding.) The results for multiple processor runs are similar. The shape of the plot confirms our claim. The "bump"s in the curves are reproducible.

For reference, the native `dgemm` routine runs for this problem size in 17.874 seconds. Thus, our best time of 33.609 seconds (at a tile size of 16) puts us at a slowdown factor of 1.88, compared to the factor of around 8 that Frens and Wise reported [12, Table 1]. The numbers for $n = 1536$ are 61.555 seconds for native `dgemm`, 96.1996 seconds for our best time, and a slowdown factor of 1.56.

Robustness of performance. To study the robustness of the performance of the various algorithms, we timed the standard and

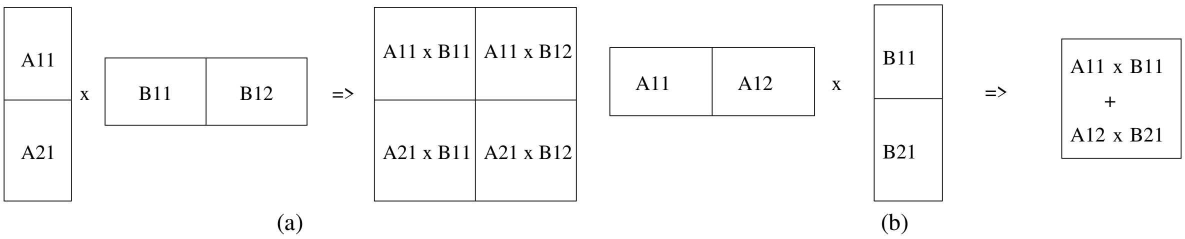


Figure 3: Handling of lean and wide matrices. (a) Lean A and wide B. (b) Wide A and lean B.

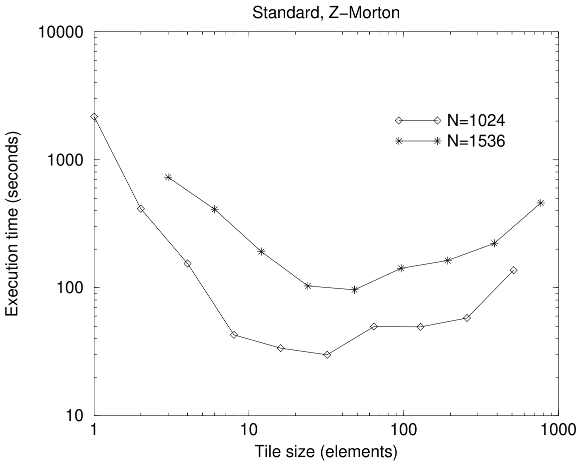


Figure 4: Effect of depth of recursive layout on performance. Standard algorithm, L_Z layout, $n = 1024$ and $n = 1536$, one processor. Note that both axes are logarithmic.

Strassen algorithms using the L_C and L_Z layouts for n in the range $[1000, 1048]$ on 1–4 processors. Figure 5 shows the results, which are unlike what we had originally expected. The standard algorithm with L_C layout exhibits large performance swings which are totally reproducible. The L_Z layout greatly reduces this variation but does not totally eliminate it. In stark contrast, Strassen's algorithm does not display such fluctuation for either layout. In neither case do we observe a radical performance loss at $n = 1024$, which is what we originally expected to observe. The fluctuations for the standard algorithm appear to be an artifact of paging, although we have not yet been able to confirm this hypothesis. We offer our explanation of the robustness of Strassen's algorithm in Section 5.1.

Relative performance of different layouts. Figure 6 shows the relative performance of the various layout functions at two problem sizes: $n = 1000$ and $n = 1200$. The figures reveal two major points. First, compared to the L_C layout, the effect of recursive layouts on the standard algorithm is dramatic, while their effect on the two fast algorithms is marginal. We offer our explanation of this effect in Section 5.1. Second, at least for these problem sizes, the performance of all the recursive layouts is approximately the same. We interpret this to mean that our implementation of the layouts is sufficiently efficient to control the addressing overheads even of L_H. An alternate explanation is that the purported benefits of, say, L_H over L_Z, do not manifest themselves until we reach larger problem sizes.

A critique of Cilk. Overall, we were favorably impressed with the capabilities of the Cilk system [4] that we used to parallelize our code. For a research system, it was quite robust. The simplicity of the language extensions made it possible for us to parallelize our codes in an afternoon. The restriction of not being able to call Cilk functions from C functions, while sound in its motivation, was the one feature that we found annoying, for a simple reason: it required to annotate several intervening C functions in a call chain into Cilk functions, which appeared to us to be spurious. This problem is avoidable by using the library version of Cilk.

The intimate connections between Cilk and `gcc`, and the limitations on linking non-Cilk libraries, are serious barriers to getting performance out of the system. In order to quantify these losses, we compiled our serial C codes with three different sets of compile and link options: (i) a baseline version compiled with the vendor `cc` (Sun's Workshop Compilers Version 4.2), with optimization level `-fast`, and linking in the native `dgemm` routine from Sun's perflib library for the leaf-level matrix multiplications; (ii) a version compiled with the vendor `cc` with optimization level `-fast`, but with our C routine instead of the native `dgemm`; and (iii) a version compiled with `gcc` version 2.7.2, with optimization level `-O3`, and with our C routine instead of the native `dgemm`. Figure 7 summarizes our measurements with these three versions for several problem sizes, algorithms, and layout functions. The results are quite uniform: the lack of native BLAS costs us a factor of 1.2–1.4, while the switch to `gcc` costs us a factor of 1.5–1.9. We find it interesting that the incremental loss in performance due to switching compilers is comparable to the loss in performance due to the non-availability of native BLAS. The single-processor Cilk running times are indistinguishable from the running times of version (iii) above, indicating an extremely efficient implementation of the Cilk runtime system.

5.1 Why the fast algorithms behave differently than the standard algorithm

Our explanation for the qualitative difference in the behavior of the fast algorithms compared to the standard algorithm is as follows. Both the Strassen and Winograd algorithms perform pre-additions, which require the allocation of quadrant-sized temporary storage, while this is not the case with the standard algorithm. Therefore, when performing the leaf-level matrix products, the standard algorithm works with tiles of the original input matrices, which have leading dimension equal to n. In contrast, every level of recursion in the fast algorithms reduces the leading dimension by a factor of approximately two, *even if we do not re-structure the matrix at the top level*. This intrinsic feature of the fast algorithms makes them less sensitive to idiosyncrasies of the memory system.

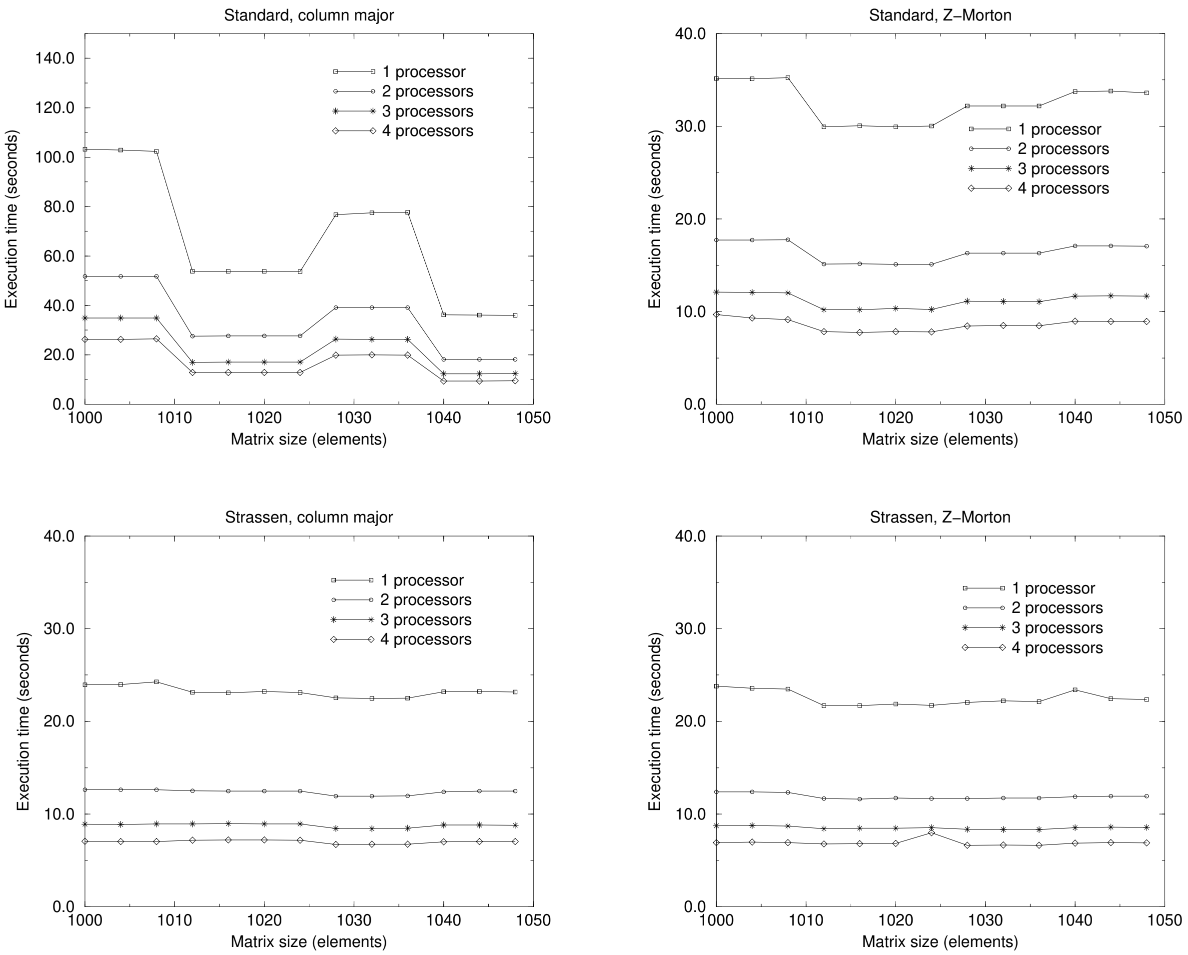


Figure 5: Performance of the standard algorithm and Strassen's algorithm, using L_C and L_Z layouts, for $n \in [1000, 1048]$, on 1–4 processors.

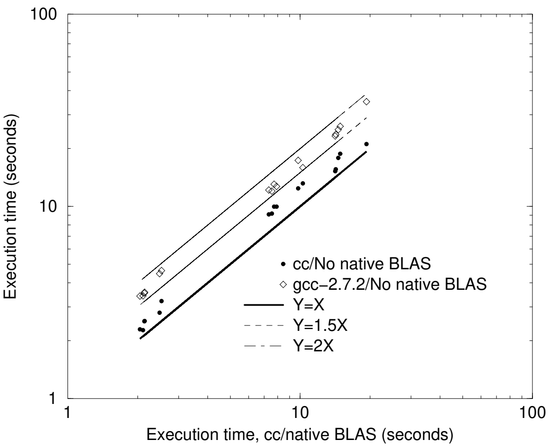


Figure 7: Overhead of gcc and of non-native BLAS.

We note one other curious feature that we have observed in the fast algorithms. If we were interested only in sequential computation, and wished to conserve space, we would intersperse recursive calls with pre- and post-additions. This version behaves more like the standard algorithm: L_Z reduces execution times by 10–20%. Of course, there is no parallelism in such a code, so it was not appropriate for this study. A systematic explanation of these curious interactions between space requirements, execution time, parallelism, and memory system architecture remains open.

6 Related work

We categorize related work into two categories: previous application of recursive array layout functions in scientific libraries, and work in the parallel systems community community related to language design and iteration space tiling for parallelism.

Scientific libraries Several projects emphasize the generation of self-tuning libraries for specific problems. We are aware of three such efforts: PHiPAC [3], ATLAS [39], and FFTW [13]. The

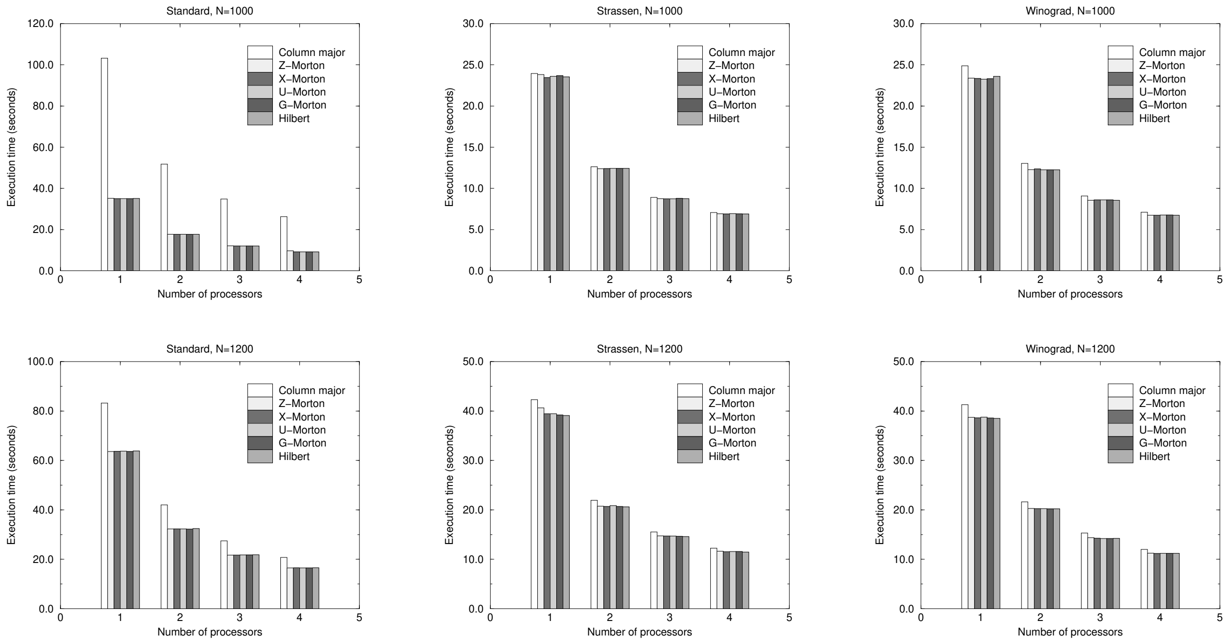


Figure 6: Comparative performance of the six layouts. The top row is for $n = 1000$, while the bottom row is for $n = 1200$. The columns from left to right are for the standard, Strassen, and Winograd algorithms.

PHiPAC project aims at producing highly tuned code for specific BLAS 3 [10] kernels such as matrix multiplication that are tiled for multiple levels of the memory hierarchy. Their approach to generating an efficient code is to explicitly search the space of possible programs, to test the performance of each candidate code by running it on the target machine, and to select the code with highest performance. It appears that the code they generate is specialized not only for a specific memory architecture but also for a specific matrix size. The ATLAS project generates code for BLAS 3 routines based on the result that all of these routines can be implemented efficiently given a fast matrix multiplication routine. The FFTW project generates fast routines for one- and multi-dimensional fast Fourier transforms. None of these projects explicitly use data restructuring, although the FFTW project recognizes their importance.

Frens and Wise [12] provide an implementation of recursive matrix multiplication. We adopted their idea of computation restructuring by recursion unfolding. They appear to carry the recursion down to the level of single array elements, which causes a dramatic loss of performance. Their reported performance is considerably lower than ours; it is difficult to tell whether this difference in performance is due to architectural improvements, to assumptions made in the algorithmic details, or to coding differences.

Gustavson [16] discusses the role of recursive control strategies in automatic variable blocking of dense linear algebra codes, and shows dramatic performance gains compared to implementations of the same routines in IBM's Engineering and Scientfic Subroutine Library (ESSL).

The application of space-filling curves is not new to parallel processing, although most of the applications of the techniques have been tailored to specific application domains [1, 20, 21, 33, 35, 38]. They have also been applied for bandwidth reduction in information theory [2], for graphics applications [14, 27], and for database applications [22]. Most of these applications have far coarser granularity than our test codes. We have shown that the overheads of these layouts can be reduced enough to make them useful for fine-grained computations.

Parallel languages and compilers. The parallel compiler literature contains much work on iteration space tiling for gaining parallelism [41] and improving cache performance [5, 40]. Carter *et al.* [6] discuss hierarchical tiling schemes for a hierarchical shared memory model. Lam, Rothberg, and Wolf [26] discuss the importance of cache optimizations for blocked algorithms. A major conclusion of their paper was that "it is beneficial to copy non-contiguous reused data into consecutive locations". Our recursive data layouts can be considered an *early binding* version of this recommendation, where the copying is done possibly as early as compile time.

The class of data-parallel languages exemplified by High Performance Fortran [25] recognizes the fact that co-location of data with processors is important for parallel performance, and provides user directives such as `align` and `distribute` to re-structure array storage into forms suitable for parallel computing. The recursive layout functions described in this paper can be fitted into this memory model using the `mapped` distribution supported in HPF 2.0. Hu *et al.*'s implementation [20] of a tree-structured N-body simulation algorithm manually incorporated L_Z within HPF in a similar manner. Support for the recursive layouts could be formally added to HPF without much trouble. The more critical question is how well the corresponding control structures (which are most naturally described using recursion and nested dynamic spawning of computations) would fit within the HPF framework.

A substantial body of work in the parallel computing literature deals with layout optimization of arrays. Representative work includes that of Mace [29] for vector machines; of various authors investigating automatic array alignment and distribution for distributed memory machines [7, 15, 23, 24]; and of Cierniak and Li [8] for DSM environments. The last paper also recognizes the importance of joint control and data optimization.

7 Conclusions

We have examined the combination of five recursive layout functions with three parallel matrix multiplication algorithms. We have demonstrated that addressing using these layout functions can be accomplished cheaply, and that these address computations can be performed implicitly by embedding them in the control structure of the algorithms. We have shown that, to realize maximum performance, such recursive layouts need to co-exist with canonical layouts, and that this interfacing can be performed efficiently. We observed no significant performance variations among the different layout functions. Finally, we observed a fundamental qualitative difference between the standard algorithm and the fast ones in terms of the benefits of recursive layouts; we attribute this difference to the algorithmic feature of pre-additions.

References

[1] I. Banicescu and S. F. Hummel. Balancing processor loads and exploiting data locality in N-body simulations. In *Proceedings of Supercomputing'95 (CD-ROM)*, San Diego, CA, Dec. 1995. Available from http://www.supercomp.org/sc95/proceedings/594_BHUM/SC95.HTM.

[2] T. Bially. Space-filling curves: Their generation and their application to bandwidth reduction. *IEEE Transactions on Information Theory*, IT-15(6):658–664, Nov. 1969.

[3] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, pages 340–347, Vienna, Austria, July 1997.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, CA, July 1995. Also see http://theory.lcs.mit.edu/~cilk.

[5] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, CA, Oct. 1994.

[6] L. Carter, J. Ferrante, and S. F. Hummel. Hierarchical tiling for improved superscalar performance. In *International Parallel Processing Symposium*, Apr. 1995.

[7] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Optimal evaluation of array expressions on massively parallel machines. *ACM Trans. Prog. Lang. Syst.*, 17(1):123–156, Jan. 1995.

[8] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 205–217, La Jolla, CA, June 1995.

[9] D. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[10] J. J. Dongarra, J. D. Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, Jan. 1990.

[11] P. C. Fischer and R. L. Probert. Efficient procedures for using matrix algorithms. In *Automata, Languages and Programming*, number 14 in Lecture Notes in Computer Science, pages 413–427. Springer-Verlag, 1974.

[12] J. D. Frens and D. S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance with source code. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 206–216, Las Vegas, NV, June 1997.

[13] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of ICASSP'98*, volume 3, page 1381, Seattle, WA, 1998. IEEE.

[14] M. F. Goodchild and A. W. Grandfield. Optimizing raster storage: an examination of four alternatives. In *Proceedings of Auto-Carto 6*, volume 1, pages 400–407, Ottawa, Oct. 1983.

[15] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, Sept. 1992. Available as technical reports UILU-ENG-92-2237 and CRHC-92-19.

[16] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, Nov. 1997.

[17] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadephia, 1996.

[18] D. Hilbert. Über stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[19] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. Comput.*, C-38(12):1612–1630, Dec. 1989.

[20] Y. C. Hu, S. L. Johnsson, and S.-H. Teng. High Performance Fortran for highly irregular problems. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–24, Las Vegas, NV, June 1997.

[21] S. F. Hummel, I. Banicescu, C.-T. Wang, and J. Wein. Load balancing and data locality via fractiling: An experimental study. In *Language, Compilers and Run-Time Systems for Scalable Computers*. Kluwer Academic Publishers, 1995.

[22] H. V. Jagadish. Linear clustering of objects with multiple attributes. In H. Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 332–342, Atlantic City, NJ, May 1990. ACM, ACM Press. Published as SIGMOD RECORD 19(2), June 1990.

[23] K. Kennedy and U. Kremer. Automatic data layout for distributed memory machines. *ACM Trans. Prog. Lang. Syst.*, 1998. To appear.

[24] K. Knobe, J. D. Lukas, and G. L. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, Feb. 1990.

[25] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. The MIT Press, Cambridge, MA, 1994.

[26] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Apr. 1991.

[27] R. Laurini. Graphical data bases built on Peano space-filling curves. In C. E. Vandoni, editor, *Proceedings of the EUROGRAPHICS'85 Conference*, pages 327–338, Amsterdam, 1985. North-Holland.

[28] C. E. Leiserson. Personal communication, Aug. 1998.

[29] M. E. Mace. *Memory Storage Patterns in Parallel Processing*. Kluwer international series in engineering and computer science. Kluwer Academic Press, Norwell, MA, 1987.

[30] M. Mano. *Digital Design*. Prentice-Hall, 1984.

[31] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properting of Hilbert space-filling curve. Technical Report CS-TR-3611, Computer Science Department, University of Maryland, College Park, MD, 1996.

[32] G. Peano. Sur une courbe qui remplit toute une aire plaine. *Mathematische Annalen*, 36:157–160, 1890.

[33] J. R. Pilkington and S. B. Baden. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):288–300, Mar. 1996.

[34] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994. ISBN 0-387-94265-3.

[35] J. P. Singh, T. Joe, J. L. Hennessy, and A. Gupta. An empirical comparison of the Kendall Square Research KSR-1 and the Stanford DASH multiprocessors. In *Proceedings of Supercomputing'93*, pages 214–225, Portland, OR, Nov. 1993.

[36] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[37] M. Thottethodi, S. Chatterjee, and A. R. Lebeck. Tuning Strassen's matrix multiplication for memory efficiency. In *Proceedings of SC98 (CD-ROM)*, Orlando, FL, Nov. 1998. Available from http://www.supercomp.org/sc98.

[38] M. S. Warren and J. K. Salmon. A parallel hashed Oct-Tree N-body algorithm. In *Proceedings of Supercomputing'93*, pages 12–21, Portland, OR, Nov. 1993.

[39] R. C. Whaley. Automatically tuned linear algebra software. In *Prceedings of SC'98*, Orlando, FL, 1998.

[40] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, June 1991.

[41] M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing'89*, pages 655–664, Reno, NV, Nov. 1989.