Cache-Efficient Matrix Transposition*

Siddhartha Chatterjee
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175, USA
sc@cs.unc.edu

Sandeep Sen
Department of Computer Science and Engineering
Indian Institute of Technology
New Delhi 110016, India
ssen@cse.iitd.ernet.in

Abstract

We investigate the memory system performance of several algorithms for transposing an $N \times N$ matrix in-place, where N is large. Specifically, we investigate the relative contributions of the data cache, the translation lookaside buffer, register tiling, and the array layout function to the overall running time of the algorithms. We use various memory models to capture and analyze the effect of various facets of cache memory architecture that guide the choice of a particular algorithm, and attempt to experimentally validate the predictions of the model. Our major conclusions are as follows: limited associativity in the mapping from main memory addresses to cache sets can significantly degrade running time; the limited number of TLB entries can easily lead to thrashing; the fanciest optimal algorithms are not competitive on real machines even at fairly large problem sizes unless cache miss penalties are quite high; low-level performance tuning "hacks", such as register tiling and array alignment, can significantly distort the effects of improved algorithms; and hierarchical non-linear layouts are inherently superior to the standard canonical layouts (such as row- or column-major) for this problem.

1 Introduction

Matrix transposition is a fundamental operation in linear algebra and in other computational primitives such as multi-dimensional Fast Fourier Transforms. This seemingly innocuous permutation problem lacks both temporal and spatial locality and is therefore tricky to implement efficiently for large matrices. Indeed, there is no temporal locality to be exploited, since each element of the matrix is accessed at most once. As far as spatial locality is concerned, the pairwise exchanges of matrix elements (i, j) and (j, i) implied by the semantics of transposition, when translated into memory addresses using a canonical row-major or column-major ordering, pairs up memory locations $ni+j$ and $nj+i$. Depending on the values of i and j, these may be either close together or far apart in terms of cache sets or virtual memory pages. Careful scheduling of these exchange operations is required to obtain any advantage of multi-word cache lines. Given the difficulty of avoiding performance problems in implementing such permutations, Gatlin and Carter [18] have named them *Murphy permutations*.

This paper uses the matrix transposition problem as a test case to evaluate six algorithms, designed to be "optimal" under various memory models, on a real modern machine. The purpose of this exercise is twofold: first, to understand how well the asymptotic predictions of the various theoretical memory models match the behavior observed in real memory hierarchies, and where the shortcomings lie; second, to analytically understand and empirically assess the relative contributions of the various components of a typical memory hierarchy (registers, data cache, translation lookaside buffer) in the running time of the operation. Our test problem to be that of transposing in-place an $N \times N$ matrix of 32-bit single-precision floating point numbers, where $N = 2^n$. Except in the final algorithm, we assume a row-major layout of the matrix in memory, as performed in C for static arrays.

The remainder of the paper is organized as follows. Section 2 reviews previous work on memory models that will be needed to understand the algorithms we study. Section 3 presents the algorithms, along with the arguments for their optimality. Section 4 presents the experimental data and its interpretation. Section 5 presents conclusions and future research directions.

*This work is supported in part by DARPA Grant DABT63-98-1-0001, NSF Grants EIA-97-26370 and CDA-95-12356, 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. Part of the work was done when the second author was a visiting faculty member in the Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, USA.

2 Related Work

Models of computation are essential for abstracting the complexity of real machines and enabling the design and analysis of algorithms. The widely-used RAM model owes its longevity and usefulness to its simplicity and robustness. While it is far removed from the complexities of any physical computing device, it successfully predicts the relative performance of algorithms based on an abstract notion of operation counts.

The RAM model assumes a flat memory address space with unit-cost access to any memory location. With the increasing use of caches in modern machines, this assumption grows less justifiable. On modern computers, the running time of a program is as much a function of operation count as of its cache reference pattern. A result of this growing divergence between model and reality is that operation count alone is not always a true predictor of the running time of a program, and manifests itself in anomalies such as a matrix multiplication algorithm demonstrating $O(n^5)$ running time instead of the expected $O(n^3)$ behavior [4].

The RAM model has been criticized for its disregard for the memory hierarchy. In particular, the difference in speeds between primary and secondary memory has become too large to ignore. The access time to disk could be 10,000 times slower than the main memory, so it is inappropriate to assign the same access cost to these memory locations. Several attempts have been made in the past ten years to incorporate this feature into the basic RAM model. Among several such models [1–4], the two-level (or external-memory) model of Aggarwal and Vitter [3] has found wide acceptance because of its relative simplicity. One of the challenges of describing a model is to achieve a a good balance between abstraction and reality, so as not to make the model too cumbersome for theoretical analysis or over-simplistic to the point of being unrealistic.

The I/O model assumes that most of the data resides on disk and has to be transferred to main memory to do any processing. Because of the tremendous difference in speeds, it ignores the cost of internal processing and counts only the number of I/Os. Floyd [14] defined a formal model and proved tight bounds on the number of I/Os required to transpose a matrix using two internal memory pages. Hong and Kung [22] extended this model and studied the I/O complexity of FFT when the internal memory size is bounded by M. Aggarwal and Vitter [3] further refined the model by incorporating an additional parameter B, the number of (contiguous) elements transferred in a single I/O operation. They gave upper and lower bounds on the number of I/Os for several fundamental problems including sorting, selection, matrix transposition, and FFT. Following their work, researchers have designed I/O-optimal algorithms for fundamental problems in graph theory [12] and computational geometry [19]. The problem of sorting has been a focus of attention, resulting in our better understanding about the I/O complexity of sorting [8].

Researchers have also modeled multiple levels of memory hierarchy. Aggarwal *et al.* [1] defined the *Hierarchical Memory Model* (HMM) that assigns a function $f(x)$ to accessing location x in the memory, where f is a monotonically increasing function. This can be regarded as a continuous analog of the multi-level hierarchy. Aggarwal *et al.* [2] added the capability of block transfer to the HMM, which enabled them to obtain faster algorithms. Alpern *et al.* [4] described the *Uniform Memory Hierarchy* (UMH) model, where the access costs differ in discrete steps. Other attempts were directed towards extracting better performance by parallel memory hierarchies [13, 34, 35], where P blocks could be transferred simultaneously.

However, the previous papers failed to capture two salient features of the cache-memory interaction: the lack of full associativity in the mapping from memory blocks to cache sets, and the lack of explicit control over data transfer between levels of the memory hierarchy. The ramifications of the previous results in the context of cache performance of an algorithm are therefore not clear. There have been attempts to improve cache performance of problems like matrix multiplication [25] and Bit reversal Permutation [9, 18] (related to FFT), but there is no general analysis of these techiques. In fact, Carter and Gatlin [9] conclude their recent paper saying

> *What is needed next is a study of "messy details" not modeled by UMH (particularly cache associativity) that are important to the performance of the remaining steps of the FFT algorithm.*

In a companion paper [31], we propose a two-level hierarchy to model the interaction between cache and main memory, that resembles the two-level I/O model but incorporates the two salient features of caches listed above. Somewhat surprisingly, the work in that paper shows that the constraint imposed by limited associativity can be tackled quite elegantly through a simple emulation scheme, so that we are able to extend the results of the I/O model to the cache model very efficiently.

Very recently, Frigo *et al.* [17] have presented an alternate strategy of algorithm design on these models which has the added advantage that explicit values of parameters related to different levels of the memory hierarchy are not required. We will discuss this model further in Section 3.4.

3 The Algorithms

We present several algorithms to transpose a square matrix in-place, and analyze their time complexity in different models. Since the computation performed by each of these algorithms is identical, the essential difference among the algorithms is the way they schedule their data exchanges. It is precisely the interaction between the schedule and the memory hierarchy that causes differences in the observed running times of the algorithms.

3.1 Algorithm 1: RAM model

Given a matrix $A = \{a_{i,j}\}$, $0 \leqslant i, j < N$, the following simple C code transposes the matrix A essentially based on the definition of transpose and does it in-place.

```
for (i = 0; i < N; i++) {
    for (j = i+1; j < N; j++) {
        tmp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = tmp;
    }
}
```

The analysis of this algorithm in the RAM model is also straightforward. The statements in the innermost loop are executed $N \cdot (N-1)/2$ times, each costing a constant number of operations, yielding a complexity of $O(N^2)$. This is considered optimal since the input consists of N^2 elements. The actual costs of the individual operations are closely related to the underlying machine architecture that is not considered important at the level of algorithm design. The goal is to study the growth rate of the running time with respect to the input size, so the RAM model assigns unit cost uniformly to all the primitive operations and ignores constant factors.

Things can change dramatically in the presence of memory hierarchy that gives rise to widely varying costs to different ranges of the memory. Consequently, the seemingly innocuous code would manifest a wide ranging behavior dependent on various parameters of the memory hierarchy. Not only do we need to reanalyze the code, but potentially redesign algorithms in the new environment with an eye towards optimality.

3.2 Algorithm 2: I/O model

The I/O model of Aggarwal and Vitter [3] to capture the interaction between the secondary memory and the main memory and study the I/O complexity of various problems. It has three parameters: the internal memory size M, and the block size B, and the input size N. The input size N of a problem is usually much larger than M and all the computations can be carried out only on elements present in the internal memory. The internal computation is not charged because of the very high cost of an I/O operation as compared to the cost of internal processing. In a single I/O, we can tranfer B elements ($M \geqslant 2B$). The goal of designing I/O algorithms is to minimize the number of I/O operations.

The problem of transposing a matrix residing in the external memory was addressed as early as Floyd[14], who designed an optimal algorithm for the case where the main memory holds two pages. This was adapted by Aggarwal and Vitter [3] in the external memory model.

What happens if we use the simple program of Section 3.1 to transpose in the I/O model? If $N > M$ (number of elements in a row/column exceeds the internal memory size), the first block in each row will be brought into the internal memory B times, corresponding to the B diagonal-symmetric elements occupying different blocks. This happens for other blocks also until the remaining matrix elements can fit in the cache. This results in $\Omega(N^2)$ I/O operations.

To reduce the number of I/O operations, we have to reschedule the operations so that they re-use elements in a block. The matrix is partitioned into (disjoint) $B \times B$ sub-matrices, where B divides N (for simplicity). Recall that B is the block size. Let $A^{r,s}$ denote the sub-matrix composed of elements $\{a_{i,j}\}, rB \leqslant i < (r+1)B$ and $sB \leqslant j < (s+1)B$, where $0 \leqslant r, s \leqslant \frac{N}{B} - 1$. Notice that each sub-matrix occupies B blocks in external memory where the elements of each row of the sub-matrix occupy B contiguous locations. However, the B blocks are separated by N elements (see Figure 1).

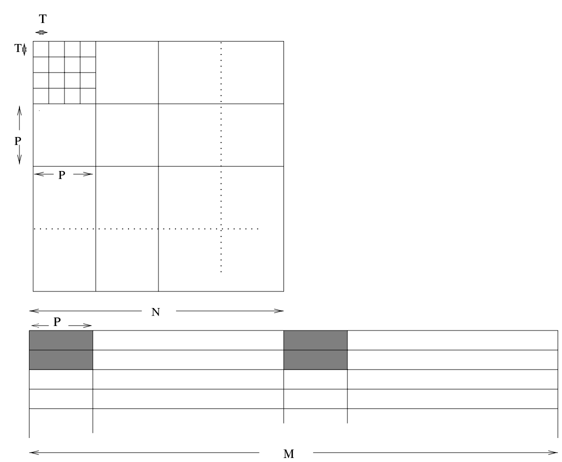

Figure 1. The row major layout of a matrix and tiles of two nested tiles of sizes B and P.

For simplicity, let us assume that the transposed matrix will be assigned to another matrix $C = A^T$ (not in-place).

Clearly,

$$C^{s,r} = (A^{r,s})^T.$$

If we assume that the internal memory is large enough to hold a sub-matrix, *i.e.*, $M > B^2$, we can accomplish the required task using the following procedure where an $n \times n$ matrix is partitioned into $B \times B$ sub-matrices.

Block-Transpose(n, B)

1. Transfer each sub-matrix $A^{r,s}$ to the internal memory using B I/O operations.
2. Perform the transpose of $A^{r,s}$ internally.
3. Transfer it to $C^{s,r}$ using B I/O operations.

The total number of I/O operations is $2\frac{N^2}{B^2} \cdot B = O\left(\frac{N^2}{B}\right)$, which is optimal. If we count the number of internal memory operations, it is also optimal, namely $O(N^2)$. To perform the transpose in-place, we will require $M \geqslant 2B^2$ to simultaneously hold $A^{r,s}$ and $A^{s,r}$. The following is a straightforward generalization that applies to $T \times T$ sub-matrices where $T \geqslant B$—this is often referred to as *tiling* and the submatrices $A^{r,s}$ are known as *tiles*.

Lemma 3.1 *For $M \geqslant T^2$, the number of I/O operations in* **Block-Transpose(N, T)** *is $O(\frac{N^2}{B})$.*

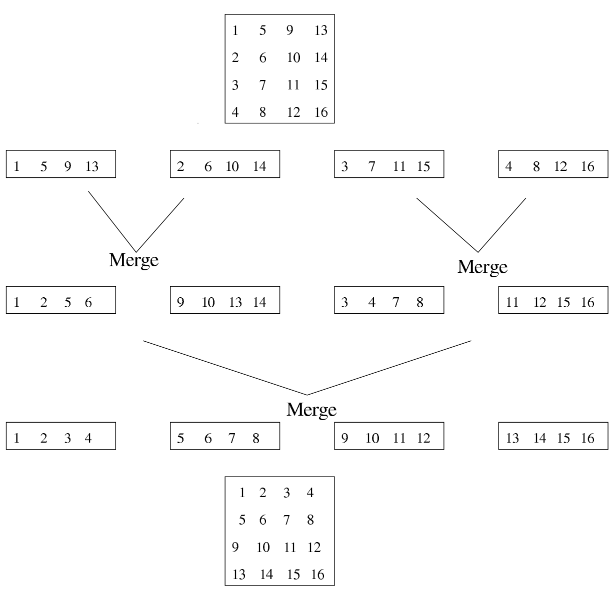

Figure 2. Transposing a 4×4 tile using 2-way merge.

The above scheme runs into problems if $M < B^2$, *i.e.*, when main memory cannot hold a $B \times B$ sub-matrix. We then resort to performing the transpose using sorting on the destination indices (see Figure 2). This is done using an $\frac{M}{B}$-way *merge* that takes $O(B\frac{\log B^2}{\log M/B})$ I/O operations. This "merge" is actually a very regular interspersing of elements rather than a comparison-based merge (see Figure 2 for details of the merge). The following result from Aggarwal and Vitter [3] (by an adaptation of Floyd's [14] proof) shows that this is an optimal scheme.

Theorem 3.1 *The number of I/O operations required to transpose an $N \times N$ matrix stored in a row-major ordering is*

$$\Theta\left(\frac{N^2}{B} \cdot \frac{\log \min\{M, 1 + N^2/B\}}{\log(1 + M/B)}\right).$$

Remark 1 This is a slightly simplified version where we have assumed that the matrix is square. Observe that for $M = \Omega(B^{1+\epsilon})$ for any fixed $\epsilon > 0$, this takes $O(N^2/B)$ I/O operations. When B is large, say $M = kB$ for some constant k, transpose takes $\Theta(N^2/B \cdot \log_k B)$ steps.

3.3 Algorithms 3 and 4: cache model

In the case of cache and main memory, the difference in access times is considerably smaller, namely a factor of 5–100. We will let L denote the *normalized cache miss latency*. The normalized cost function assigns a cost of 1 for accessing cache and L otherwise. This way, we will account for the computation in cache also. In the context of the cache, we will continue to use M for cache size and B for block size. The block-size B is much smaller (about 4–8 elements as opposed to 1000) and referred to as the *cache line*. Therefore our *cache model* [31] has four parameters, namely N, M, B and L, one more than the I/O model. Although we have chosen M for both the main memory size (in the context of I/O model) and cache size (in the cache model), the reader should think of M as the size of the *faster* memory.

A significant distinction between the two models is the degree of *associativity* available. In the I/O model, we can transfer any block from the external memory to any block in the internal memory. In contrast, the main-memory blocks are mapped into the cache sets using a *fixed* mapping function. Typically a number of low-end address bits are used to map a main-memory location into the cache. This maps contiguous main-memory locations to contiguous locations in cache (modulo M). We assume a *direct-mapped* cache.

A further difference in the way the two models behave is the lack of explicit control on the cache locations. The cache is not visible to the programmer (not even at the assembly level). When a program starts running, an *image* (copy) of the the block containing a memory reference is brought into the corresponding cache set (unless it is already present), and it continues to be there till it is evicted by another block that is mapped to the same cache set. In other words, a cache set contains the latest memory block referenced that is mapped to this set.

These differences would frustrate any efforts to naively map an I/O algorithm to the cache, given that we neither have the control nor the flexibility of the I/O model. Sen and Chatterjee[31] establish a useful relationship between the I/O model and the cache-model using a very simple emulation.

Theorem 3.2 ([31]) *If an algorithm A in the I/O model uses T block-transfers and I processing time, then the algorithm can be executed in the cache model in $O(I + (L + B) \cdot T)$ steps. The memory requirement is an additional $M/B + 2$ blocks beyond the external-memory algorithm.*

The idea behind the emulation is to use a memory-resident array *Buf* of the same size as cache (M) that mimics the role of the internal memory of the I/O algorithm. Since *Buf* consists of contiguous locations, there are no interference misses between these locations. With careful use of copying involving locations other than *Buf*, the theorem can be proved using amortized analysis. The constant in the emulation overhead is small (about 2).

The term $O(B \cdot T)$ is subsumed by $O(I)$ if computation is done on at least a constant fraction of the elements in the block transferred by the I/O algorithm. This is usually the case for efficient I/O algorithms. We will call such I/O algorithms *block-efficient*. The algorithms for sorting, FFT, matrix transpose and matrix multiplication described in Aggarwal and Vitter [3] are block-efficient.

Corollary 3.3 *A block-efficient I/O algorithm that uses T block transfers and I processing can be executed in the cache model in $O(I + L \cdot T)$ steps.*

If we implement **Block-Transpose()** directly, there will be several problems caused mainly by the limited associativity in cache. All the blocks in in a $B \times B$ tile $A^{r,s}$ may be mapped to the same cache set. As Gatlin and Carter[18] argue, this is not merely a theoretical possibility but a very likely situation if N is a multiple of the cache-size M. This would cause thrashing between the contending blocks leading to $\Omega(B^2)$ misses per tile instead of te desired B misses, amounting to a total of N^2 misses.

This phenomenon can be avoided by using the emulation. In conjunction with Corollary 3.3, the I/O-efficient scheme for matrix transpose yields the following result for transposing in the cache model.

Theorem 3.4 *An $N \times N$ matrix can be transposed in the cache model in*

$$O\left(N^2 + L \cdot \left(\frac{N^2}{B} \cdot \frac{\log \min\{M, 1 + N^2/B\}}{\log(1 + M/B)}\right)\right)$$

steps. For $M = \Omega(B^{1+\epsilon})$, where $\epsilon > 0$, this is optimal.

Remark 2 In the context of matrix transpose, the procedure implied by the Emulation Theorem is analogous to the **COBRA** procedure of Gatlin and Carter [18] for Bit Reversal permutation. Our description can be viewed as a formal derivation starting from the I/O model.

Most architectures have a hierarchy of cache-memory levels. Here, we only discuss the effect of the *Translation Lookahead Buffer* (TLB), which is a cache used for storing virtual to physical address translations. While this is also a cache, it has some special characteristics: it is a small number of entries, the span of each entry is large, and it is highly associative (often fully associative). Let B_T and k represent the block size and the number of TLB entries respectively. In most machines, $B_T >> k$. Since the blocks of the tile $A^{r,s}$ are separated by more than B_T, we will encounter B TLB misses per tile, namely the same as the number of cache misses. If TLB misses are more expensive, then this component would dominate.

From Remark 1, it follows that the best we can do with respect to TLB misses is $\Omega(\frac{N}{B_T} \cdot \log_k B_T)$, which requires $\log_k B_T$ passes through the $N \times N$ matrix. Consequently this will increase the number of cache misses by a factor of $\log_k B_T$ and the trade-off can be evaluated only on the basis of the actual values of the cache miss and the TLB miss penalties.

Optimizing multiple levels of cache appears to be a hard problem theoretically. Carter and Gatlin[9, 18] address a restricted problem in the context of Bit Reversal Permutation, namely, how to minimize number of TLB misses given that we want to keep the cache misses optimal (one round trip to cache per matrix element). If we assume that $\sqrt{C} \leqslant B_T$, then we can use a tile size of $\sqrt{C}$ (Lemma 3.1) and we can bound the number of TLB misses to $\sqrt{C}$ per $\sqrt{C} \times \sqrt{C}$ tile or $\sqrt{C} \cdot \frac{N^2}{C}$ total TLB misses which is less than N/B misses for $\sqrt{C} > B$.

In practical terms, the theoretical discussions above motivate two algorithms. It is clear that we need to copy matrix blocks to and from contiguous storage in order to avoid catastrophic conflict miss effects. The flip side of copying is that it increases the number of instructions and memory references. Unlike the case in matrix multiplication [26], we cannot amortize the copying cost over multiple uses of a block. We therefore implemented two versions of **Block-Transpose**, with different amounts of copying. Figure 3 illustrates these variants. The first variant, which we call *half copying*, increases the number of data movement steps

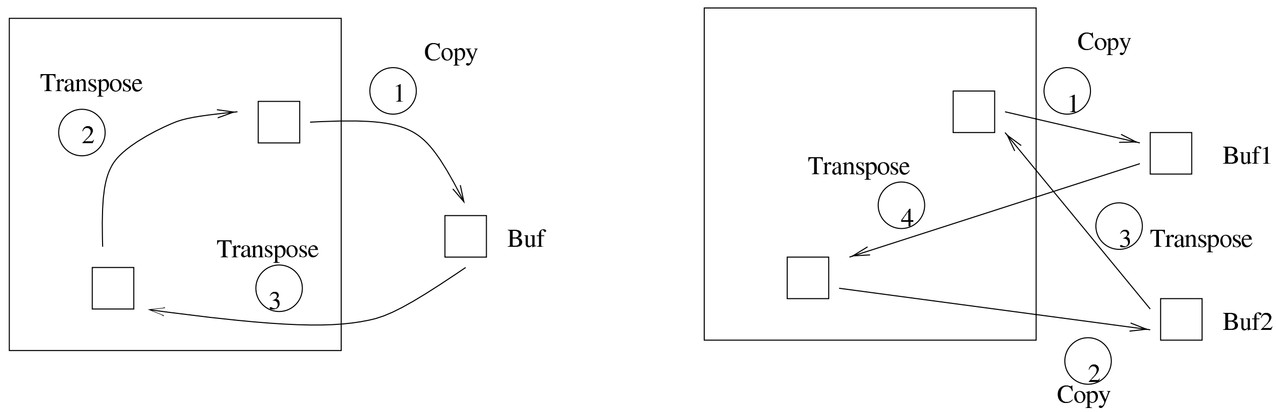

Figure 3. Implementing Block-Transpose with half copying (left) and full copying (right).

from 2 to 3, while reducing the number of conflict misses. The second variant, which we call *full copying*, increases the number of data movement steps to 4, but completely eliminates conflict misses. Both these variants use auxiliary storage that occupies $\Theta(B^2)$ space.

3.4 Algorithm 5: Cache-oblivious

Frigo *et al.* [17] present an alternate strategy of algorithm design which has the added advantage that explicit values of parameters related to different levels of the memory hierarchy are not required. They call such algorithms "cache-oblivious" because they contain no variables dependent on hardware parameters that need to be tuned to achieve optimality, they are asymptotically optimal in terms of work and data movement in a "tall ideal cache" model (which reasonably models a fully associative data cache, but not, for example, a TLB). The basic idea is to use a divide-and-conquer strategy to divide the problem into successively smaller subproblems. Intuitively, the subproblems will fit in cache once they are small enough. Frigo *et al.* give the following algorithm for transposing a rectangular $m \times n$ matrix A into an $n \times m$ matrix B.[1]

> If $n \geqslant m$, we partition
>
> $$A = (A_1 A_2), B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}.$$
>
> Then, we recursively execute $\textsc{Transpose}(A_1, B_1)$ and $\textsc{Transpose}(A_2, B_2)$. If $m > n$, we divide matrix A horizontally and matrix B vertically and likewise perform two transpositions recursively.

They prove the following optimality result for their algorithm.

[1] The algorithm is easily specialized for the in-place square case.

Theorem 3.5 ([17]) *The cache-oblivious matrix-transpose algorithm involves $O(mn)$ work and incurs $O(1 + mn/L)$ cache misses for an $m \times n$ matrix and a cache line size of L elements, which is asymptotically optimal.*

3.5 Algorithm 6: Non-linear array layout

The final algorithm we use is similar to the cache-oblivious algorithm in control structure, but uses a different layout of the matrix. This is based on the observation that canonical array layouts (such as row-major) 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 can reduce program performance in several ways. First, it may reduce or even nullify the effectiveness of multi-word cache lines. Such low spatial locality can usually be corrected by appropriate loop transformations (such as interchange, reversal, or skewing) when such transformations are legal [6]. but this does not help in the matrix transposition example. Second, for large matrix sizes, it may even reduce the effectiveness of translation lookaside buffers (TLBs), because the dilation effect extends to virtual memory pages [5, 33]. Finally, it may cause cache misses due to self-interference even when a tiled loop repeatedly accesses a small tile in the array index space, because the canonical layout depends on the matrix size rather than the tile size. Such interference misses are a complicated and non-smooth function of the array size, the tile size, and the cache parameters [16]. These considerations lead us to investigate other, nonlinear, array layout functions.

The nonlinear layout function we use has been variously described as being based either on quadtrees [15] or on space-filling curves [20, 28, 30]. This layout is known in parallel computing as the *Morton ordering* and has been used for load balancing purposes [7, 23, 24, 29, 32, 36]. Fig-

```
static void tr1(int src, int num)

{
  if (num==1) {
/* base case: single tile
   exchange with loop nest */
  }
  else {
    int nn1 = num/4;
    tr1(src      ,nn1);
    tr2(src+  nn1,src+2*nn1,nn1);
    tr1(src+3*nn1,nn1);
  }
}

static void tr2(int src, int dst, int num)

{
  if (num==1) {
/* base case: single tile
   exchange with loop nest */
  }
  else {
    int nn1 = num/4;
    tr2(src      ,dst      ,nn1);
    tr2(src+  nn1,dst+2*nn1,nn1);
    tr2(src+2*nn1,dst+  nn1,nn1);
    tr2(src+3*nn1,dst+3*nn1,nn1);
  }
}

int
main(int argc, char *argv[])

{
  tr1(0, nblks*nblks);
}
```

Figure 4. Code skeleton for transpose algorithm with non-linear layout.

ure 5 illustrates this layout. Our interest, however, is in exploiting the benefits of such orderings for multi-level memory hierarchies.

Morton ordering has the following operational interpretation. Divide the original matrix into four quadrants, and lay out these submatrices in memory in the order NW, NE, SW, SE. A $k_R \times k_C$ submatrix with $k_R > t_R$ and $k_C > t_C$ is laid out recursively using the Morton ordering; a $t_R \times t_C$ tile is laid out using the L_F-ordering.

To formally define this layout function, we require t_R and t_C to simultaneously satisfy

$$\frac{m}{t_R} = \frac{n}{t_C} = 2^d \tag{1}$$

for some positive integer d. We define

$$L_T(t_i, t_j; t_R, t_C) = t_R \cdot t_C \cdot \mathbb{M}(t_i, t_j)$$

where $\mathbb{M}(i,j)$ is the integer whose binary representation is the bitwise interleaving of the binary representations of i and j. Then,

$$\begin{aligned} L_{MO}(i,j;m,n,t_R,t_C) &= t_R \cdot t_C \cdot \mathbb{M}(t_i,t_j) \\ &+ L_{CM}(f_i,f_j;t_R,t_C). \end{aligned} \tag{2}$$

See Chatterjee *et al.* [10, 11] for further details and implementation issues for this layout.

Like the cache-oblivious algorithm, this algorithm also uses recursion to divide the problem into smaller subproblems until it reaches an architecture-specific tile size, where it performs the exchanges. The code is shown in Figure 4. There are two differences between this algorithm and the cache-oblivious one. First, the layout function of the matrix is Morton-ordered rather than row-major. This makes every tile contiguous in memory and cache space, and eliminates self-interference misses when tiles are transposed. Second, the recursion is terminated at an architecture-specific tile size rather than down to single elements as in a cache-oblivious scheme.

4 Experimental Results

All our experiments were run on a 300 MHz UltraSPARC-II system. The L1 data cache is direct-mapped, with 32-byte blocks and a capacity of 16 KB. The L2 data cache is direct-mapped, with 64 byte blocks and a capacity of 2 MB. The system has 512 MB of RAM. The VM page size is 8 KB, and the data TLB is fully associative with 64 entries. The system runs SunOS 5.6, and we used SUN's Workshop Compilers 4.2. In addition to timing runs, we also performed cache simulations using the FAST-CACHE and CPROF tools [21, 27].

Figure 6 shows the running times of the various algorithms for a number of different problem sizes and tile sizes. From these numbers, Algorithms 6 and 3 emerge the fastest, with Algorithm 4 coming in a close third. Algorithms 2 and 5 are in the next group. with Algorithm 1 bringing up the rear. There is a 5x improvement between Algorithms 1 and 6, and a 4x improvement between Algorithms 1 and 3.

In order to understand the behavior of the various algorithms, we need to look at their memory system behavior. Figure 7 summarizes this information for the six algorithms. (For brevity, we include data for a single tile size only.) The following points emerge from this data.

1. The number of data references varies greatly among the algorithms. Algorithms 1, 5, and 6 perform the absolute minimum number of data references necessary. The extra number of data references in Algorithms 2, 3, and 4 are as predicted by the analysis.

2. Algorithms 3, 4, and 5, by virtue of working on submatrices, reduce TLB misses somewhat. Algorithm 2 was specifically designed to reduce TLB misses, and the reduction is dramatically clear. (Note, however, that this gain comes at the expense of many more data references.) The TLB misses of Algorithm 6 are even smaller, reflecting the fact that VM pages now hold submatrices rather than rows or columns of the original matrix. Correlating this with running times reveals that TLB misses are quite significant on this platform.

3. The data cache misses of Algorithm 4 are fewer in number than those of Algorithm 3, but this gain is offset by the extra memory references of Algorithm 4.

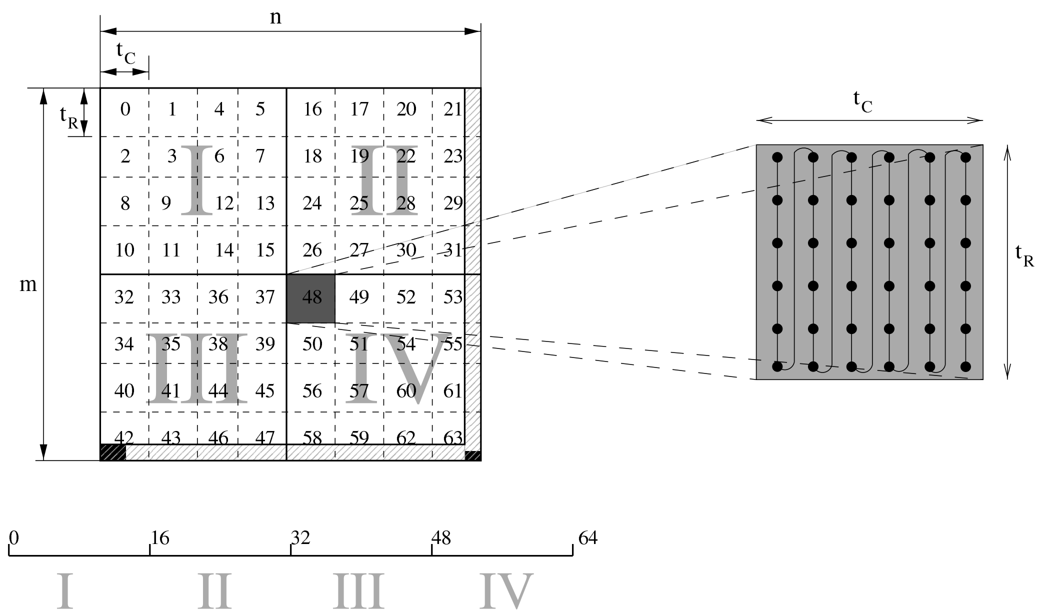

Figure 5. The Morton layout function, with $t_R \times t_C$ tiles. Each tile is internally organized in column-major manner.

Running time (seconds), block size = 2^5						
$\log_2 N$	Alg.1	Alg. 2	Alg. 3	Alg. 4	Alg. 5	Alg. 6
10	0.21	0.10	0.06	0.06	0.08	0.03
11	0.86	0.49	0.39	0.35	0.45	0.14
12	3.37	1.63	1.05	1.14	2.16	0.54
13	13.56	6.38	4.55	4.99	6.69	2.13

Running time (seconds), block size = 2^6						
$\log_2 N$	Alg.1	Alg. 2	Alg. 3	Alg. 4	Alg. 5	Alg. 6
10	0.13	0.08	0.06	0.05	0.08	0.03
11	0.85	0.42	0.34	0.28	0.45	0.13
12	3.38	1.58	0.89	0.97	1.97	0.52
13	13.51	5.99	3.58	3.91	7.00	2.09

Running time (seconds), block size = 2^7						
$\log_2 N$	Alg.1	Alg. 2	Alg. 3	Alg. 4	Alg. 5	Alg. 6
10	0.14	0.12	0.05	0.05	0.09	0.03
11	0.87	0.42	0.36	0.24	0.47	0.20
12	3.36	1.46	0.85	0.88	2.03	0.59
13	13.46	5.74	3.12	3.35	6.86	2.35

Figure 6. Running times of the six algorithms, for various matrix sizes and block sizes. Alg. 1 is the naive algorithm. Alg. 2 is the "merge" algorithm of the I/O model. Alg. 3 is the "half-copying" algorithm. Alg. 4 is the "full-copying" algorithm. Alg. 5 is the cache-oblivious algorithm. Alg. 6 is the algorithm with Morton layout of the matrix. Algorithms 1 and 5 do not depend on the block size parameter.

$N = 2^{10}, B = 2^6$			
Alg.	Data refs	L1 misses	TLB misses
1	2,097,032	589,795	258,679
2	6,293,417	575,517	2,126
3	3,164,040	722,120	16,556
4	4,196,232	275,550	16,475
5	2,097,008	131,226	8,096
6	2,096,642	150,716	535

$N = 2^{11}, B = 2^6$			
Alg.	Data refs	L1 misses	TLB misses
1	8,386,434	2,362,002	2,096,165
2	25,168,901	2,294,658	12,781
3	12,617,602	2,944,721	134,019
4	16,779,138	1,170,003	133,501
5	8,386,434	923,295	134,951
6	8,387,043	607,907	2,091

$N = 2^{12}, B = 2^6$			
Alg.	Data refs	L1 misses	TLB misses
1	33,548,161	9,453,724	8,391,854
2	100,671,883	9,169,771	69,124
3	50,399,119	11,841,230	544,012
4	67,110,785	4,804,808	542,814
5	33,548,161	7,101,600	516,749
6	33,552,371	2,442,151	8,323

$N = 2^{13}, B = 2^6$			
Alg.	Data refs	L1 misses	TLB misses
1	134,203,282	37,826,712	33,572,154
2	402,685,862	36,641,659	276,762
3	201,459,602	47,480,885	2,175,318
4	268,437,378	19,493,808	2,172,819
5	134,203,282	56,158,873	2,009,964
6	134,221,843	9,790,139	33,267

Figure 7. Number of data references, number of L1 data misses, and number of data TLB misses for the various algorithms, for a number of different problem sizes and a block size of 2^6.

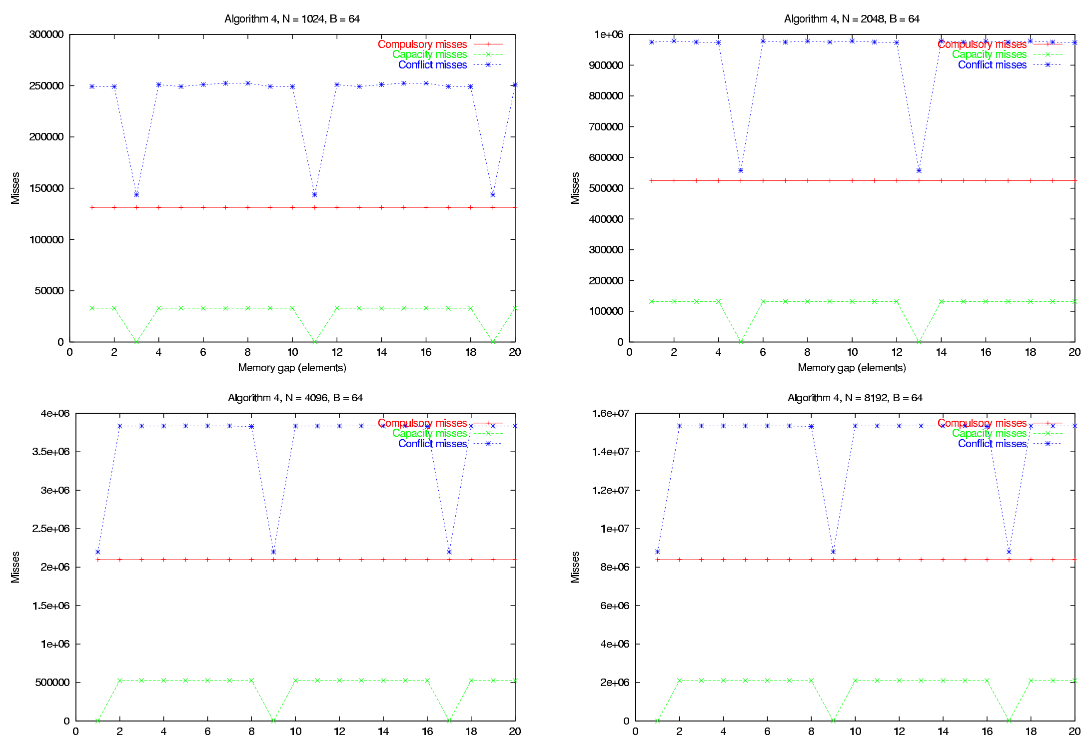

Figure 8. Misses as a function of the gap in memory between the array being transposed and the buffer used to copy blocks.

The relative importance of the two factors depends on the cache miss penalty. Back-of-the-envelope calculations reveal that Algorithm 4 would outperform Algorithm 3 if the cache miss latency was greater than 5–10 cycles. With the growing disparity between processor and memory speeds, this may soon be the regime of operation. A similar comment holds for Algorithm 2 as TLB miss penalties increase.

4. The cache-oblivious algorithm has a low data reference count, but high cache and TLB misses. Some of this stems from carrying the recursion down to single elements. We have experimented with a version of this algorithm where we terminate the recursion at the same block size as the other algorithms (thus making it "cache-aware"). This improves its running time to the extent that it beats Algorithm 2. The remainder of the mismatch between theory and practice may be a result of the fully-associative model used in the analysis of this algorithm.

We conclude this section by discussing two low-level system effects that were essential to obtaining these results. The first is register tiling. In the tiled implementation of **Block-Transpose**, writing the loop as follows:

```
for (i = 0; i < BLK_SIZE_ELTS; i++) {
  for (j = 0; j < BLK_SIZE_ELTS; j++) {
    buf1[j][i] = A[u+i][v+j];
  }
```

results in catastrophic conflict misses between `A` and `buf1`, as also noted by Gatlin and Carter [18]. To avoid this effect, we need to avoid interleaving the accesses to the two arrays, and instead use registers to buffer an entire cache line worth of data, thus.

```
for (i = 0; i < BLK_SIZE_ELTS; i++) {
  for (j = 0; j < BLK_SIZE_ELTS;
       j += HW_BLK_SIZE_ELTS) {
    t0 = A[u+i][v+j+0];
    t1 = A[u+i][v+j+1];
    t2 = A[u+i][v+j+2];
    t3 = A[u+i][v+j+3];
    t4 = A[u+i][v+j+4];
    t5 = A[u+i][v+j+5];
    t6 = A[u+i][v+j+6];
    t7 = A[u+i][v+j+7];
    buf1[j+0][i] = t0;
    buf1[j+1][i] = t1;
    buf1[j+2][i] = t2;
    buf1[j+3][i] = t3;
    buf1[j+4][i] = t4;
    buf1[j+5][i] = t5;
    buf1[j+6][i] = t6;
    buf1[j+7][i] = t7;
  }
}
```

One would hope that the compiler would not undo this intended blocking of reads and writes, but such is not the case at high optimization levels. The Sun compiler with the `-fast` optimization option ignores the desired buffering and proceeds to interleave the reads and writes. We therefore had to turn the optimization level down to `-xO2` for Algorithms 2, 3, and 4, to observe the desired behavior in terms of cache misses and running times.

The other low-level effect involved the gap in memory between the starting addresses of arrays `A` and `buf1`. Figure 8 shows the variation in the three different kinds of misses as this gap varies, for Algorithm 4 at different problem sizes and a block size of 64. The variation in misses is a result of conlflict misses between the two arrays, which are almost always catastrophic, as an *a posteriori* analysis reveals. The effect is periodic with a period of eight elements (this being the number of floating-point values in a cache line), with the minima being observed at different values of the gap at different problem sizes. More importantly, almost all (seven out of eight) choices of gaps are bad, so that choosing a random gap size will not solve the problem. The results presented above were taken with the memory gap set to a value that minimizes misses. This phenomenon affects Algorithms 2, 3, and 4.

5 Conclusions

We have investigated six different algorithms for the problem of matrix transposition. While these algorithms perform the same algebraic operation, they schedule the operations in very different ways, placing different loads on the various components of the memory system. We have tried to correlate the predicted performance of the algorithms with their observed behavior, and have tried to explain the differences.

We note that the asymptotic analysis of the algorithms matches their cache miss behaviors better than their running times, even for problem sizes that should be "large" by any reasonable measure. It is clear that tighter analysis of running times is needed if one is to make any meaningful predictions about running times, as there is a fine balance to be struck between the total number of data references and the number of misses. As a corollary, the notion of optimality in the various memory models does not necessarily model reality.

The relative performance of the algorithms depends critically upon the cache miss penalty. Some of the algorithms that were not the front-runners in the test environment may yet be relevant if the miss penalty increases by virture of the ever-growing gap between processor and memory speeds.

Finally, alternative data layouts for matrices, such as the Morton layout, have superior properties in a hierarchical memory setting, and should be seriously considered for many dense matrix computations.

References

[1] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of ACM Symposium on Theory of Computing*, pages 305–314, 1987.

[2] A. Aggarwal, A. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proceedings of IEEE Foundations of Computer Science*, pages 204–216, 1987.

[3] A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(5):1118–1127, 1988.

[4] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2):72–109, 1994.

[5] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. G. Steinberg, and K. Yelick. Empirical evaluation of the CRAY-T3D: A compiler perspective. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 320–331, Santa Margherita Ligure, Italy, June 1995.

[6] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, 1993. ISBN 0-7923-9318-X.

[7] 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.

[8] R. Barve, E. Grove, and J. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):109–118, 1997. A preliminary version appeared in SPAA 96.

[9] L. Carter and K. Gatlin. Towards an optimal bit-reversal permutation program. In *Proceeding of IEEE Foundations of Computer Science*, 1998.

[10] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, pages 444–453, Rhodes, Greece, June 1999.

[11] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, Saint-Malo, France, June 1999.

[12] Y. Chiang, M. Goodrich, E. Grove, R. Tamassia, D. Vengroff, and J. Vitter. External memory graph algorithms. In *Proceedings of the ACM-SIAM Symposium of Discrete Algorithms*, pages 139–149, 1995.

[13] T. H. Cormen, T. Sundquist, and L. F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM Journal of Computing*, 28(1):105–136, 1999.

[14] R. Floyd. Permuting information in idealized two-level storage. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 105–109. Plenum Press, New York, NY, 1972.

[15] 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.

[16] C. Fricker, O. Temam, and W. Jalby. Influence of cross-interference on blocked loops: A case study with matrix-vector multiply. *ACM Trans. Prog. Lang. Syst.*, 17(4):561–575, July 1995.

[17] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of IEEE Foundations of Computer Science*, 1999. To appear.

[18] K. S. Gatlin and L. Carter. Memory hierarchy considerations for fast transpose and bit-reversals. In *Proceedings of HPCS 5*, Orlando, FL, Jan. 1999. IEEE.

[19] M. Goodrich, J. Tsay, D. Vengroff, and J. Vitter. External memory computational geometry. In *Proceeding of IEEE Foundations of Computer Science*, pages 714–723, 1993.

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

[21] M. D. Hill, J. R. Larus, A. R. Lebeck, M. Talluri, and D. A. Wood. Wisconsin architectural research tool set. *Computer Architecture News*, 21(4):8–10, August 1993.

[22] J. Hong and H. Kung. I/O complexity: The red blue pebble game. In *Proceedings of ACM Symposium on Theory of Computing*, 1981.

[23] 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.

[24] 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.

[25] M. Lam, E. Rothberg, and M. 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 63–71, 1991.

[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] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE COMPUTER*, 27(10):15–26, October 1994.

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

[29] 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.

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

[31] S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. Submitted for publication, July 1999.

[32] 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.

[33] M. R. Swanson, L. Stoller, and J. Carter. Increasing TLB reach using superpages backed by shadow memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 204–213, June 1998.

[34] J. Vitter and M. Nodine. Large scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing*, 17:107–114, 1993.

[35] J. Vitter and E. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2):110–147, 1994.

[36] 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.