AHNENTAFEL INDEXING INTO MORTON-ORDERED ARRAYS, or MATRIX LOCALITY FOR FREE[⋆]

David S. Wise[⋆⋆]

Indiana University

Abstract. Definitions for the uniform representation of d-dimensional matrices serially in Morton-order (or Z-order) support both their use with cartesian indices, and their divide-and-conquer manipulation as quaternary trees. In the latter case, d-dimensional arrays are accessed as 2^d-ary trees. This data structure is important because, at once, it relaxes serious problems of locality and latency, and the tree helps schedule multiprocessing. It enables algorithms that avoid cache misses and page faults at all levels in hierarchical memory, independently of a specific runtime environment.

This paper gathers the properties of Morton order and its mappings to other indexings, and outlines for compiler support of it. Statistics elsewhere show that the new ordering and block algorithms achieve high flop rates and, indirectly, parallelism without any low-level tuning.

CCS Categories and subject descriptors: E.1 [Data Structures]: Arrays; D.3.2 [Programming Languages]: Language Classifications–concurrent, distributed and parallel languages; applicative languages; D.4.2 [Operating Systems]: Storage management–storage hierarchies; E.2 [Data Storage Representations]: contiguous representations.
General Term: Design.
Additional Key Words and Phrases: caching, paging, compilers, quadtree matrices.

1 INTRODUCTION

Maybe we've not been representing them efficiently for some time. Matrix problems have been fodder for higher-level languages from the beginning [1], and row- or column-major representations for matrices are universally assumed. Both use the same space, both still survive.

But maybe both are archaic perspectives on matrix structure, which might best be represented using a third convention. Architecture has developed quite a bit since we had to pack scalars into small memory. With hierarchical—rather than flat—memory, only the faster memory is precious; with distributed processing instructions on local registers are far faster than those touching remote memory. And, of course, multiprocessing on many cheap processors demands less crosstalk than code for single threading on uniprocessors with unshared

⋆ Copyright ©2000 to Springer-Verlag, with rights reserved for anyone to make digital or hard copies of part or all of this work for personal or classroom use, provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full Springer citation on the first page. Rights are similarly reserved for any library to share a hard copy through interlibrary loan.

⋆⋆ Supported, in part, by the National Science Foundation under grants numbered CDA93–03189 and CCR-9711269. Author's address: Computer Science Dept, 215 Lindley Hall, Bloomington, IN 47405–7104, USA. `dswise@cs.indiana.edu`

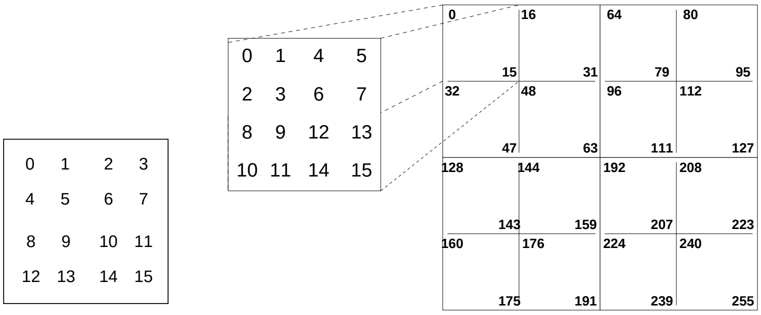

Figure 1. Row-major indexing of a 4×4 matrix, and analogous Morton indexing.

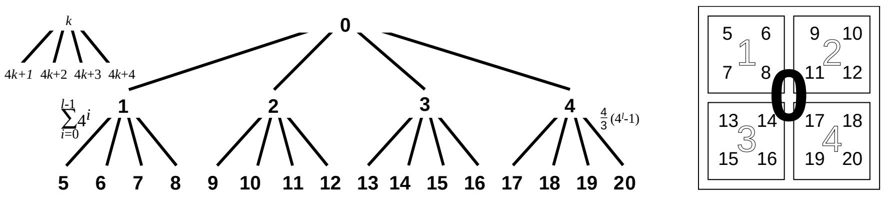

Figure 2. Level-order indexing of the order-4 quadtree and its submatrices.

memory. Address space, itself, has grown so big that chunks of it are extremely cheap, when mapped to virtual memory and never touched. (Intel's IA-64 processor has three levels of cache, two on board.) But fast cache, local to each processor, remains dear, and, perhaps, row-major and column-major are exactly wrong for it.

This paper enhances Morton-order (also called Z-order) storage for matrices. Consistent with the conventional sequential storage of vectors, it also provides for the usual cartesian indexing into matrices (row, column indices). It extends to higher dimensional matrices. That is, we can provide cartesian indexing for "dusty decks" while we write new divide-and-conquer and recursive-descent codes for block algorithms on the same structures. HASKELL and ML could share arrays with FORTRAN and C. Thus, parallel processing becomes accessible at a very high level in decomposing a problem. Best of all, the content of any blocks is addressed sequentially and blocks' sizes vary naturally (they undulate) to fit the chunks transferred between levels of the memory hierarchy.

2 BASIC DEFINITIONS

Morton presented his ordering in 1966 to index frames in a geodetic data base [11]. He defines the indexing of the "units" in a two-dimensional array much as in Figure 1, and he points out the truncated indices available for enveloping blocks (subtrees), similar to Figure 3. Finally, he points out the conversion to and from cartesian indexing available through bit interleaving.

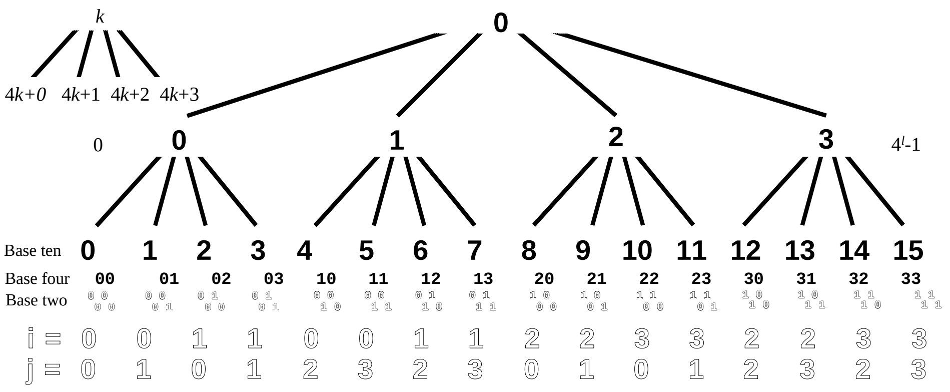
k
4k+0 4k+1 4k+2 4k+3
0
0 1 2 3
4^l-1
Base ten 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Base four 00 01 02 03 10 11 12 13 20 21 22 23 30 31 32 33
Base two
i = 0 0 1 1 0 0 1 1 2 2 3 3 2 2 3 3
j = 0 1 0 1 2 3 2 3 0 1 0 1 2 3 2 3

Figure 3. Morton indexing of the order-4 quadtree.

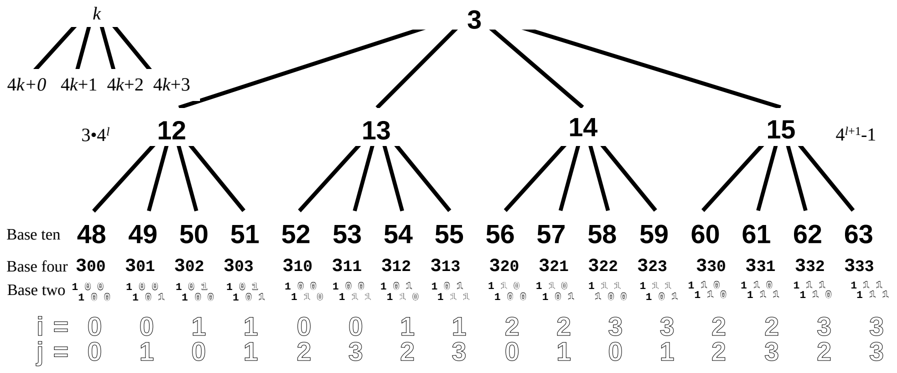
k
4k+0 4k+1 4k+2 4k+3
3
$3 \cdot 4^l$
12 13 14 15
4^{l+1}-1
Base ten 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
Base four 300 301 302 303 310 311 312 313 320 321 322 323 330 331 332 333
Base two
i = 0 0 1 1 0 0 1 1 2 2 3 3 2 2 3 3
j = 0 1 0 1 2 3 2 3 0 1 0 1 2 3 2 3

Figure 4. Ahnentafel indexing of the order-4 quadtree.

The definitions later focus on two-dimensional arrays: matrices. The early ones are general: a d-dimensional array is decomposed as a 2^d-ary tree.

2.1 ARRAYS

Definition 1 *In the following* $\mathrm{m} = 2^d$ *is the degree of the tree appropriate to the dimension* d.

If the maximal order in any dimension is n then the tree has maximal level $\lceil \lg n \rceil$. Use Figures 2, 3, and 4 in reading the following definitions.

Definition 2 *A complete array has* level-order *index* 0. *A subarray (block) at level-order index* i *is either a scalar, or it is composed of* m *subarrays, with level-order indices* $mi+1, mi+2, \ldots, mi+m$.

Definition 3 *The root of an array has* Morton-order *index* 0. *A subarray (block) at Morton-order index* i *is either a unit (scalar), or it is composed of* m *subarrays, with indices* $mi+0, mi+1, \ldots, mi+(m-1)$ *at the next level.*

Theorem 1. *The difference between the level-order index of a block at level l in an array and its Morton-order index is* $(m^l - 1)/(m-1)$.

The difference is the number of nonterminal nodes above level l. Since each level is indexed by its zero-based scheme, it is necessary also to know the level and a Morton index, to identify a specific node.

Ahnentafel indices are immensely useful for identifying blocks at all levels [17]. Algorithms that use recursive-descent (divide-and-conquer) to descend to a block of arbitrary size, or to return the index of a selected block, need only this single index to identify any subtree. Treat them only as identification numbers because there are gaps in the sequence between levels. Conversions among Ahnentafel indices, cartesian indices, Morton order, and level order are easy.

Ahnentafel indices come to us from genealogists who invented them for encoding one's pedigree as a binary, family tree. This generalization to m-ary trees is new.

Definition 4 [4] *A complete array has* Ahnentafel *index* $m-1$. *A subarray (block) at Ahnentafel index* a *is either a scalar, or it is composed of* m *subarrays, with indices* $ma+0, ma+1, \dots, ma+(m-1)$.

Theorem 2. *The nodes at level l have Ahnentafel indices from* $(m-1)m^l$ *to* $m^{l+1}-1$. *The gap in indices between level $l-1$ and level l is* $m^l(m-2)+1$.

Theorem 3. *The level of a node with Ahentafel index a is* $l = \lfloor \log_m a \rfloor = \lfloor \log_m \frac{a}{m-1} \rfloor$. *The difference between the Ahnentafel index and the level-order index is* $(m^{l+1}(m-2)+1)/(m-1)$. *The difference between the Ahnentafel index and the Morton-order index is* $(m-1)m^l$.

The gaps between levels in Ahnentafel indexing of quadtrees do not arise in binary trees. The strong similarity between level-order and Ahnentafel indexing in this common case perhaps explains why the latter has been often overlooked. For instance, Knuth's level-order indexing, based at one, is off-by-one relative to Definition 2 [10, p. 401], but coincides with Ahnentafel indexing on binary trees.

2.2 MATRICES

Hereafter, we assume $d = 2$ for matrices; so $m = 4$. In all the figures, the cartesian indices of the leaves appear in outlined font below the tree.

Corollary 1. *The difference between level-order index of a matrix block at level l and its Morton-order index is* $(4^l - 1)/3$.

Corollary 2. *The gap in Ahnentafel indices between level $l-1$ and l is* $2^{2l+1}+1$.

Corollary 3. *The difference between the Ahnentafel index of a submatrix at level l and its level-order index is* $(2^{2l+3}+1)/3$.
The difference between its Ahnentafel and its Morton-order index is $3 \cdot 4^l$.

Definition 5 *Let* w *be the number of bits in a (short) word.*
Each $\mathrm{q_k}$ *is a modulo-4 digit (or* quat*).*
Each q_k *is alternatively expressed as* $q_k = 2i_k + j_k$ *where* $\mathrm{i_k}$ *and* $\mathrm{j_k}$ *are* bits.

Cartesian indices have w bits; Morton indices (and, later, dilated integers) have $2w$ bits.

Theorem 4. [11] *The Morton index*$\sum_{k=0}^{w-1} q_k 4^k = 2\sum_{k=0}^{w-1} i_k 4^k + \sum_{k=0}^{w-1} j_k 4^k$ *corresponds to the cartesian indices: row* $\sum_{k=0}^{w-1} i_k 2^k$ *and column* $\sum_{k=0}^{w-1} j_k 2^k$

The proof is by simple induction on w; doubling the order of a matrix introduces two high-order bits. The quats, read in order of descending subscripts, select a path from the root to the node, as in Figure 4. For example, if $i = 4 = 10_4$ and $j = 8 = 20_4$ in Figure 1, the Morton index is $200_4 + 1000_4 = 3000_4 = 96_{10}$.

Corollary 4. *Let* $l = \lfloor \log_4 a \rfloor$. *The Ahnentafel index,* $a = \sum_{k=0}^{l} q_k 4^k$ *corresponds to the Morton index* $\sum_{k=0}^{l-1} q_k 4^k$.

The bits, $\{i_k\}$, are the odd-numbered bits in the Morton index, and the $\{j_k\}$ are just the even-numbered bits; and, excluding Corollary 3's two high-order **1** bits, coincident with those in an Ahnentafel index. This is Morton's bit interleaving of cartesian indices.

Code to convert from cartesian indices to a Morton index by shuffling bits—or the inverse conversion that deals out the bits—can be slow. Fortunately, as the next section shows, most conversions can be elided.

Definition 6 *The integer* $\overrightarrow{\mathrm{b}} = \sum_{k=0}^{w-1} 4^k$, *here labeled* `evenBits` *in C, and is the constant* `0x55555555`*). Similarly,* $\overleftarrow{\mathrm{b}} = 2\overrightarrow{b}$ *is called* `oddBits`, `0xaaaaaaaa`*).*

In C code $\overrightarrow{b}$ is a very important constant available independently of w as `((unsigned int)-1)/3)`. Masking a Morton index with $\overrightarrow{b}$ or $\overleftarrow{b}$ extracts the bits of the column and row cartesian indices. Morton describes how to use these to obtain indices of neighbors. Their easy identification makes the indexing attractive for graphics in two dimensions and for spatial data bases in three.

It is remarkable how often these basic properties of Morton ordering have been reintroduced in different contexts [3, 7, 9, 12, 16]. Samet gives an excellent history [13].

When first introduced, it might appear that Morton indexing is poor for large arrays that are not square or whose size is not a power of two, because in those cases its gaps seem to waste space. With the valid elements justified to the north and west, the "wasted" space lands in the south and east. It can be viewed as padding, perhaps all zero elements. However, the perceived waste is merely *address* space. In hierarchical memory only its margin will ever be swapped into cache. That is, the "wastage" exists only within the logical/physical addressing of swapping disk; little valuable, fast memory is lost.

With these orders defined and the various index translations among them understood, we can summarize the programmer's view of Morton order:

– The elements of a matrix (an array) are mapped onto memory in Morton order.
– Row and column traversals can still be handled. See the next section.
– If necessary with cartesian indexing, restrict blocking to submatrices implicit in the quadtree (those with Ahnentafel indices.)
– If data is associated also with nonterminal blocks, store it in level order.
– Use Ahnentafel indices to control recursive-descent algorithms [17].
– Ahnentafel indices are monotonic across any row or column, so bounds checking remains available via masking with `evenBits` or `oddBits`.

An early context for implementation of this design is HASKELL whose higher-dimension array aggregates (also, array comprehensions) do not imply any particular internal representation; that is, no code depends on a particular ordering. Moreover, asynchronous, parallel algorithms are implict in that style. Not accidentally, implementations of HASKELL and other functional languages do a good job implementing recursion in preference to iteration. Nevertheless, the algebra of indices implemented in the strength reduction and loop unrolling of FORTRAN and C compilers does not yet have an analog under recursion. Morton order, with its own algebra of indexing and recursion unfolding, offers this leverage and opens access to new scientific algorithms with functional style.

3 CARTESIAN INDEXING AND MORTON ORDERING

The following techniques for cartesian indexing of Morton-order arrays seem to be hardly known, a fact that is unfortunate because they make the structure useful for blocking matrices even if only used with cartesian indexing. In particular, this section newly shows how to compile row and column traversals (of blocks at any level of the tree) with the reductions in operator strength associated with optimizing compilers.

3.1 DILATED INTEGERS

The algebra of dilated integers is surprisingly old. Tocher outlined it in 1954 and under similar constraints to those again motivating us: non-flat memory with access time dependent on locality, and nearby information more rapidly accessible [15, p. 53–55]. But how the size of the memory has changed! Tocher needed fast access into a $32 \times 32 \times 32$ boolean array stored on a drum (4kB!).

Schrack shows how to effect efficient cartesian indexing from row i and column j indices into Morton-order matrices [14]. The trick is to represent i and j as dilated integers, with information stored only in every other bit.

Definition 7 *The* even-dilated representation *of* $j = \sum_{k=0}^{w-1} j_k 2^k$ *is* $\sum_{k=0}^{w-1} j_k 4^k$, *denoted* $\overrightarrow{\jmath}$. *The* odd-dilated representation *of* $i = \sum_{k=0}^{w-1} i_k 2^k$ *is* $2\overrightarrow{\imath}$, *denoted* $\overleftarrow{\imath}$.
The arrows suggest the justification of the meaningful bits in either dilated representation. For example, the right arrow suggests rightmost Bit 0 and even kin.

Theorem 5. *A matrix of m rows and n columns should be allocated as a sequential block of* $\overleftarrow{m-1} + \overrightarrow{n-1} + 1$ *scalar addresses.*

This value is, of course, the Morton index of the southeast-most element of the matrix, plus one for (the northwest-most) one whose Morton index is 0. Not all of that sequence need be active; undefined data at the idle addresses will remain resident in the lowest level of the memory hierarchy. Only data in the active addresses will ever migrate to cache.

Theorem 6. [14] *If "$\equiv$" is read as semantic equivalence and "$=$" denotes equality on integer representations, then for unsigned integers*

$$(\overrightarrow{\imath} = \overrightarrow{\jmath}) \equiv (i = j) \equiv (\overleftarrow{\imath} = \overleftarrow{\jmath});$$

$$(\overrightarrow{\imath} < \overrightarrow{\jmath}) \equiv (i < j) \equiv (\overleftarrow{\imath} < \overleftarrow{\jmath}).$$

So comparison of dilated integers is effected by the same processor commands as those for ordinary integers.

Definition 8 *The following theorems apply to w-bit 2's-complement integers. The infix operator $\wedge$ indicates bitwise conjunction; $\vee$ denotes bitwise disjunction.*
Instead of representing i and j internally, represent them as $\overleftarrow{\imath}$ and $\overrightarrow{\jmath}$.

Theorem 7. *The Morton index for the $\langle i, j\rangle^{\mathrm{th}}$ element of a matrix is $\overleftarrow{\imath} \vee \overrightarrow{\jmath}$, equivalently $\overleftarrow{\imath} + \overrightarrow{\jmath}$.*

Often (*normalized* dilated integers [14]) addition will be used instead of disjunction to associate at compile time with an adjacent addition.

Addition and subtraction of dilated integers can be performed with a couple of minor instructions.

Definition 9 *Addition ($\overset{\rightarrow}{+}, \overset{\leftarrow}{+}$) and subtraction ($\overset{\rightarrow}{-}, \overset{\leftarrow}{-}$) of even- and odd-dilated integers:*

$$\overrightarrow{\jmath}\overset{\rightarrow}{-}\overrightarrow{n} = \overrightarrow{j-n}; \qquad \overleftarrow{\imath}\overset{\leftarrow}{-}\overleftarrow{n} = \overleftarrow{i-n}.$$

$$\overrightarrow{\jmath}\overset{\rightarrow}{+}\overrightarrow{n} = \overrightarrow{j+n}; \qquad \overleftarrow{\imath}\overset{\leftarrow}{+}\overleftarrow{n} = \overleftarrow{i+n}.$$

Theorem 8. *Subtraction, addition, constant addition, and shifts on dilated integers:*

$$\overrightarrow{\jmath}\overset{\rightarrow}{-}\overrightarrow{n} = (\overrightarrow{\jmath} - \overrightarrow{n}) \wedge \overrightarrow{b}; \qquad \overleftarrow{\imath}\overset{\leftarrow}{-}\overleftarrow{n} = (\overleftarrow{\imath} - \overleftarrow{n}) \wedge \overleftarrow{b}; \qquad [14]$$

$$\overrightarrow{\jmath}\overset{\rightarrow}{+}\overrightarrow{n} = (\overrightarrow{\jmath} + \overleftarrow{b} + \overrightarrow{n}) \wedge \overrightarrow{b}; \qquad \overleftarrow{\imath}\overset{\leftarrow}{+}\overleftarrow{n} = (\overleftarrow{\imath} + \overrightarrow{b} + \overleftarrow{n}) \wedge \overleftarrow{b}; \qquad [14]$$

$$\overrightarrow{\imath}\overset{\rightarrow}{+}\overrightarrow{c} = \overrightarrow{\imath}\overset{\rightarrow}{-}\overrightarrow{(-c)}; \qquad \overleftarrow{\imath}\overset{\leftarrow}{+}\overleftarrow{c} = \overleftarrow{\imath}\overset{\leftarrow}{-}\overleftarrow{(-c)};$$

$$\overrightarrow{b} = \overrightarrow{(-1)}; \qquad \overleftarrow{b} = \overleftarrow{-1};$$

$$\overrightarrow{i{<<}k} = \overrightarrow{\imath}{<<}(2k); \qquad \overleftarrow{i{<<}k} = \overleftarrow{\imath}{<<}(2k);$$

$$\overrightarrow{i{>>}k} = \overrightarrow{\imath}{>>}(2k); \qquad \overleftarrow{i{>>}k} = \overleftarrow{\imath}{>>}(2k).$$

Theorem 8 and 6 suggest that the C loop

```
for (int i=0; i<n; i++){...}
```

be compiled to `int nn=`$\overleftarrow{n}$ and

```
for (int ii=0; ii<nn; ii=(ii-oddBits)&oddBits ){...}
```

So, if i and j are not represented literally as integers, but translated by the compiler to their images, $\overleftarrow{\imath}$ and $\overrightarrow{\jmath}$, the resulting object code can be just a simple homomorphic image of what the programmer expected. Code like the following source might be demanded from the programmer, but it would be better introduced via a transformation by the helpful compiler:

```
#define evenBits ((unsigned int) -1)/3)
#define oddBits  (evenBits <<1)
#define  oddIncrement(i)  (i= ((i -  oddBits) &  oddBits))
#define evenIncrement(j)  (j= ((j - evenBits) & evenBits))
...
   for     (i = 0; i< rowCountOdd ;  oddIncrement(i))
     for   (j = 0; j< colCountEven; evenIncrement(j))
       for (k = 0; k< pCountEven  ; evenIncrement(k))
          c[i + j] += a[i + k] * b[j + 2*k];
```

The index computations of $\overrightarrow{k}$ and $\overleftarrow{k}$ in the innermost loop above reduce to three RISC instructions, plus two to sum the matrix-element addresses. Although fast and constant time, this is still half-again what is available from column-major representation. With integer / floating processors tuned to column-major this difference may once have mattered but now it doesn't, with register operations so fast relative to memory access. It becomes important, though, for compilers to provide this arithmetic as loop control.

For three and higher dimensions, it seems best to implement arithmetic only for one type of dilated integer, and to translate other representations from it.

Dilated integers simplify input and output of Morton-ordered matrices in human-readable raster order [3]. Such convenience is not computationally significant because I/O delays dominate that indexing, but it may be politically important just to make this matrix representation accessible to the programmers who need to experiment with it.

3.2 SPACE AND BOUNDS

Theorem 5 tells how much address space an $m \times n$ matrix occupies, usually more than the mn positions containing data. As mentioned above and observed in experiments, if the difference, $\overleftarrow{m-1} + \overrightarrow{n-1} + 1 - mn$, is large then most of it will never move into faster memory. Moreover, the larger the difference, the more remote (and cheaper) will be the bulk of the addresses allocated. The size of that excess space for a rectangular matrix depends the number of elements and also on the aspect ratio between the census of rows and columns.

In block algorithms, cache hits make it better to iterate through Morton-order sequentially. For instance, if `b` is the initial index of a block from matrix A of size `n`$=4^p$ that is to be zeroed, it is better to initialize it with a single, localized loop: `for (int i=0; i<n; i++) A[b+i]=0;` than to use column-major traversal.

Bounds checking on each row and column is elegant. First of all, Theorem 6 provides a fast check of either a Morton or an Ahnentafel index, `a`, against predilated row and column bounds:

```
if ( (a &oddBits)<rowCountOdd && (a &evenBits)<colCountEven ) ...
```

For Ahnentafel indices especially, the compiler can do even more. It will precompute two vectors of bounds on a row or column index at each level of the quadtree. The first, `rowBound[]`, is a bound on the perimeter of the matrix, to preclude access to southern and eastern padding. The second, `rowDense[]`, contains bounds on interior Ahnentafel indices that are north and west of this perimeter which, once passed, obviate the need for further bounds checking at any subtree / subblock. Allocate two vectors of size h to contain row bounds on the Ahnentafel indices at each level, where h is the height of the tree. For both, the h^{th} entry is $\overleftarrow{r}$ where r is the number of rows. Thereafter,

```
rowBound[level-1] = rowBound[level]>>2;
rowDense[level-1] = (rowDense[level]$\overleftarrow{-}\overleftarrow{1}$) >>2;
```

With c columns the column bounds are computed from $\overrightarrow{c}$ similarly, as even dilated integers; we usually test row and column limits simultaneously.

One might not be overly precise on Ahnentafel bounds at the perimeter. Because extra space is often allocated to the south and east anyway, it has been observed to be cheaper to round the actual bounds on the matrix up to, say, the next multiple of eight and then to treat the margins as “active” padding. For a recursive block algorithm, the alternative is to detect blocks of size four, two, and one where there will be too few operations; with the smallest block at order eight, operations on it can be dispatched as unconditional, straight-line, superscalar code. That is, we choose to treat padding in small marginal blocks as sentinels, initialized so each can participate in the gross algorithm without affecting its net result. Typically this is zero to the south or east, and the identity matrix to the southeast:

```
for (int jj=0; jj<n; jj= (jj-evenBits)&evenbits ) A[3*jj]=1;
```

The compiler also needs to use the algebra of dilated integers from Section 3.1 to unfold recursions and to unroll loops effectively; it becomes another kind of reduction in the strength of indexing.

Finally, symmetric matrices and matrix transpose are also easy with Morton or Ahnentafel indices. If `m` is either kind of index, then the index of its reflected element or (untransposed) block is quickly computed by exchanging its even and odd bits in a dosido: `( (m &evenBits) <<1) + ( (m &oddBits) >>1)`

4 CONCLUSION

We have already demonstrated vast improvements using both the cartesian and Ahentafel indexing schemes described here.

A good quadtree algorithm uses recursive descent on Ahnentafel indices, so there is no need to preselect a block size to fit one—or any—level of cache; the algorithm simply accelerates when a block fits [8]. None of the indices, themselves, need be stacked. Each can be shifted right to effect a stack pop or incremented to refer to a sibling. Still, innermost recursions need to be unfolded [2], just as the C compiler unrolls loops, in order to obtain straight-line code for superscalar processors. See [18] for more details; all the speed improvements are due to good locality [5].

The formalism presented here shows how to implement Morton-order matrices, with efficient algorithms for the group of dilated integers for the program that uses cartesian indexing, or with Ahnentafel indexing for the one that uses recursive descent on quadtrees. We have built a propotype compiler to translate C programs using row-major matrices and cartesian indices to Morton-order using dilated indices. We had already demonstrated the ease of tree-wise scheduling parallel processors in [7], and we continue to search for similar quadtree algorithms [17, 6].

It remains to close the gap between these efforts: on the one hand to improve compilers to optimize indexing on Morton-ordered matrices (*e.g.* unfolding Ahnentafel–controlled recursions and unrolling dilated-integer–controlled loops). And, on the other hand, to exercise Morton-order matrices under both styles, comparing performance of both families of algorithms by comingling them in a single environment as two libraries of interchangeable modules. Our underlying goal remains to uncover better algorithms that use more locality and balanced parallelism to solve matrix problems faster.

Acknowledgement: Thanks to Christian Weiss for helpful comments and special thanks to Hanan Samet who provided the early references.

References

1. J. Backus. The history of FORTRAN I, II, and III. In R. L. Wexelblat (ed.), *History of Program. Languages*, New York, Academic Press (1981), 25–45. Also preprinted in *SIGPLAN Not.* **13**, 8 (1978 August), 165–180.
2. R. M. Burstall & J. Darlington. A transformation system for developing recursive programs. *J.ACM* **24**, 1 (1997 January), 44–67.
3. F. W. Burton, V. J. Kollias, J. G. Kollias. Real-time raster-to-quadtree and quadtree-to-raster conversion algorithms with modest storage requirements. *Angew. Informatik* **4** (1986), 170–174.
4. H. G. Cragon A historical note on binary tree. *SIGARCH Comput. Archit. News* **18**, 4 (1990 December), 3.
5. D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, & T. von Eicken. LogP: a practical model of parallel computation. *Commun. ACM* **39**, 11 (1996 November), 78–85.
6. J. Frens. *Matrix Factorization Using a Block-Recursive Structure and Block-Recursive Algorithms.* PhD dissertation, Indiana University, Bloomington (in progress).
7. J. Frens & D. S. Wise. Auto-blocking matrix multiplication, or tracking BLAS3 performance from source code. *Proc. 1997 ACM Symp. on Principles and Practice of Parallel Program., SIGPLAN Not.* **32**, 7 (1997 July), 206–216.
8. M. Frigo, C. E. Leiserson, H. Prokop, & S. Ramachandran. Cache-oblivious algorithms, Extended abstract. Lab for Computer Science, M.I.T., Cambridge, MA (1999 May). `http://supertech.lcs.mit.edu/cilk/papers/abstracts/FrigoLePr99.html`
9. Y. C. Hu, S. L. Johnsson & S.H. Teng. High performance Fortran for highly irregular problems. *Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program., SIGPLAN Not.* **32**, 7 (1977 July), 13–24.
10. D. E. Knuth. *The Art of Computer Programming* **I,** *Fundamental Algorithms* (3rd ed.), Reading, MA: Addison-Wesley, (1997).
11. G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Ottawa, Ontario: IBM Ltd. (1966 March 1).
12. J. A. Orenstein & T. H. Merrett. A class of data structures for associative searching. *Proc. 3rd ACM SIGACT–SIGMOD Symp. on Princ. of Database Systems* (1984), 181–190.
13. H. Samet. *The Design and Analysis of Spatial Data Structures* Reading, MA: Addison-Wesley, (1990), §2.7.
14. G. Schrack. Finding neighbors of equal size in linear quadtrees and octrees in constant time. *CVGIP: Image Underst.* **55**, 3 (1992 May), 221-230.
15. K. D. Tocher. The application of automatic computers to sampling experiments. *J. Roy. Statist. Soc. Ser. B* **16**, 1 (1954), 39–61.
16. M. S. Warren. & J. K. Salmon. A parallel hashed oct-tree N-body problem. *Proc. Supercomputing '93.* Los Alamitos, CA: IEEE Computer Society Press (1993), 12–21.
17. D. S. Wise. Undulant block elimination and integer-preserving matrix inversion. *S-ci. Comput. Program.* **33**, 1 (1999 January), 29–85. `http://www.cs.indiana.edu/ftp/techreports/TR418.html`
18. D. S. Wise & J. Frens. Morton-order matrices deserve compilers' support. Technical Report 533, Computer Science Dept, Indiana University (1999 November). `http://www.cs.indiana.edu/ftp/techreports/TR533.html`