

# Using Space-filling Curves for Computation Reordering

Guohua Jin    John Mellor-Crummey  
Department of Computer Science  
Rice University  
6100 South Main Street, Houston, TX 77005

## Abstract

Space-filling curves have been widely used in mathematics and to transform multi-dimensional problems into one-dimensional forms. For scientific applications, ordering data or computation along space-filling curves can be useful for exploiting locality when partitioning onto parallel systems or when restructuring to exploit memory hierarchy. In this paper, we present an efficient approach for enumerating points or mapping to points along space-filling curves. We introduce a new table specification, *position specification*, for mapping to points along space-filling curves. We discuss how this framework can be used for reordering computation. Experiments on three modern microprocessor based platforms show that our algorithm is up to a factor of 72 faster for enumerating points along a curve and up to a factor of 11 faster for mapping to points on the curve than a previous byte-oriented, non-recursive implementation.

## 1 Introduction

Space-filling curves have been used in a variety of fields including mathematics, algorithms, geographical information systems, image processing, databases, circuit design, cryptology, and scientific computing [8, 23, 1, 29, 5, 2, 19, 26, 10, 20, 21, 16, 14]. Salmon and Warren used space-filling curve for partitioning and reordering computation in N-body simulations for improving data locality of both their in-core and out-core implementations [26]. Challacombe used space-filling curves to efficiently partition a sparse matrix computation onto a parallel system [10]. Mellor-Crummey et. al [20, 21] and Hu et. al [14] investigated using space-filling curves for data and computation reordering to improve memory hierarchy utilization for irregular applications.

Two main research areas on space-filling curves are *curve traversal* and *curve indexing*. Traversing a space-filling curve is to enumerate points along

the curve. Curve indexing refers to mapping coordinate position of a point in a multi-dimensional space into its traversal position along a curve and back. Bially presented algorithms for converting back and forth between a point in an  $n$ -dimensional cube and a number representing a position along a space-filling curve [4]. He described the algorithm in a diagrammatic form and gave a numerical procedure to construct the diagrams. Butz proposed an algorithm for generating a Hilbert curve by mapping a point in a one-dimensional space into a point in an  $n$ -dimensional space by using circular shift and exclusive-or operations on bytes [9]. Although one can generate a Hilbert curve in an arbitrary dimensional space by following Bially's or Butz's description, their algorithms are inefficient.

Many researchers have studied recursive algorithms for efficiently generating space-filling curves in two and three dimensional spaces [25, 24, 13, 11, 28]. Bartholdi et al. [3] described a vertex-labeling method to generate algorithms for manipulating a Hilbert curve in 2D space and its application to the generation of 2D Sierpiński and Peano curves. Prusinkiewicz al. [24] described how to use the formalism of L-systems for expressing and drawing *FASS* curves — curves that are *space-Filling*, *self-Avoiding*, *Simple* and *self-Similar*. More recently, Breinholt and Schierz gave an algorithm for generating a 2D Hilbert space-filling curve using integer operations and recursion [6]. However, none of these recursive curve generation algorithms can be easily adapted to higher-dimensional spaces.

Several researchers have studied curve indexing in two- and three- dimensional spaces [12, 17, 18]. Fisher described a bit-serial algorithm for generating coordinates of any point along a Hilbert curve in a unit square given the traversal position of the point, a Hilbert process and an inverse function, called Berthil process [12]. A state transition table *ptab* and two other tables *ctab* and *ttab* were used to determine orientation and output values in different iterations of the processes. Liu and Schrack presented algo-

rithms for *encoding* (from coordinates of a point to its traversal position) and *decoding* (from traversal position of a point to its coordinates) a 2D and 3D Hilbert curve [17, 18].

Our approach differs from other techniques described in the literature in two ways. First, we use a different curve representation and two types of specification tables – a *movement specification table* and a *position specification table*. Our curve representation is simple and general. The movement specification table indicates the direction of each move along a curve. The position specification tables record information about coordinate and traversal positions of each point along a curve. Second, we use a table-driven turtle traversal algorithm to enumerate points in a multi-dimensional space along a space-filling curve and table-driven indexing algorithms for transforming coordinate position of a point into its traversal position along a space-filling curve and back. The tables are pre-generated based on both geometric and arithmetic properties of the curve. This enables us to generate space-filling curves in arbitrarily high-dimensional spaces much more efficiently than previous approaches. We evaluated the overall performance for traversing and indexing three different space-filling curves and compared the results using Hilbert curve with the results from a fast implementation of Butz’s byte-oriented non-recursive algorithm by Moore [22]. Experimental results on three modern microprocessor-based platforms show that our curve generation algorithm performs up to a factor of 72 faster on curve traversal and up to a factor of 11 faster on curve indexing than the previous byte-oriented non-recursive implementation. Traversing and indexing a Hilbert curve is more efficient than the other two space-filling curves in most test cases.

The rest of the paper is organized as follows. Section 2 briefly describes SFCGen, a framework for generating space-filling curves. Section 3 presents a new table specification for indexing. Section 4 describes table-driven algorithms for traversing and indexing a space-filling curve. Section 5 discusses how the framework for curve generation can be used for ordering computation. Section 6 reports experimental results and Section 7 presents our conclusions.

## 2 SFCGen: A Framework for Curve Generation

SFCGen is a general framework for generating a space-filling curve in arbitrarily high dimensional space [15]. In SFCGen, we use a general but simple curve representation for space-filling curves. A space-filling

Table 1: Movement specification table for Hilbert curves.

Current Level	Next Level			
ne	en, n	ne, e	ne, s	ws, $\perp$
sw	ws, s	sw, w	sw, n	en, $\perp$
ws	sw, w	ws, s	ws, e	ne, $\perp$
en	ne, e	en, n	en, w	sw, $\perp$

curve at any recursion level is represented by its base level approximation, called *primitive curve*. Each primitive curve is described as a sequence of critical moves. To represent an  $n$ -dimensional Morton, Hilbert, or Peano primitive curve, we use  $n$  moves  $m_1m_2\dots m_n$ , where  $m_j(1 \leq j \leq n)$  describes the move from the  $k^{j-1}$ -th node to its immediate successor node and  $k$  is the refinement factor of the curve. For example, the four Hilbert primitive curves<sup>1</sup> in Figure 1(a) are **se**, **ne**, **en**, **wn** curves. “s”, “n”, “w”, and “e” stand for south, north, west, and east moves. With this representation, a full set of  $n$ -dimensional Hilbert, Morton, or Peano primitive curves includes  $2^n n!$  different primitive curves.

To traverse a space-filling curve efficiently, we use a movement specification table. A movement specification table consists of a vector of table rows. Each table row recursively defines a space-filling curve. Each entry in a table row is a pair. The first element in the pair specifies a table row representing a primitive curve at the next level of recursion. The second element in the pair specifies the movement direction after finishing the aforementioned primitive curve. In our movement specification table, each curve is represented by its approximation at the base level. For example, the curve in Figure 1(b) is represented by “ne” in the table. The bold gray curve shows the approximated primitive curve. Table 1 shows the movement specification table for generating 2D Hilbert curves. It contains only four table rows for the primitive curves that are needed to create the curve in Figure 1(b). The “Next Level” portion of the table row should be read left to right. “ $\perp$ ” in the last table entry of each row represents no movement.

Specification tables for different space-filling curves are precomputed automatically based on geometric and arithmetic properties of the curves. To build the tables efficiently, we use a demand-driven table generation algorithm. The algorithm only creates curve specifications that are actually needed during the process of curve generation. This approach also significantly reduces the table sizes which improves

<sup>1</sup>There are eight Hilbert directed primitive curves in two dimensions.

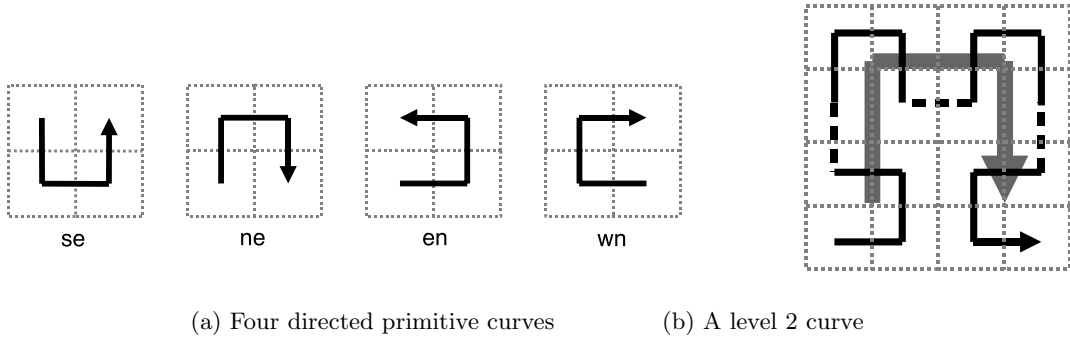


Figure 1: 2D Hilbert curves.

performance of the traversing algorithm.

### 3 Position Specification

A position specification table describes position of each subcurve at the next level in the curve from which it is refined. We use two different position specification tables, *c2i* and *i2c*, which contain information about coordinate position and traversal position of a subcurve in the enclosing curve at the next level. Like the movement specification table, both the *c2i* and *i2c* tables contain a “Current Level” column and multiple “NextLevel” columns for defining curve refinement. However, the information recorded in these two types of tables is quite different. The movement specification table specifies moves that connect a sequence of subcurves, while the position specification tables record position of each subcurve. Table 2 and 3 show the *c2i* and *i2c* tables for a 2D Peano curve. In the *c2i* table, the coordinate position of each subcurve is its column number under the “Next Level” section. The traversal position is recorded in the second element of each table entry. In contrast, the *i2c* table represents the coordinate position by the second element of each table entry and the traversal position of each subcurve by its column number.

For example, the second entry of the fourth table row in Table 2 shows that a “se” curve at the next level is in coordinate position (0, 1) or 1 since it is in the second column and its traversal position along the “ne” curve at the current level is 5 as shown in Figure 2. On the other hand, the second table entry of the fourth table row in Table 3 shows that subcurve “nw” at the next level is in coordinate position (1, 0) or 3 since the second element of the table entry is 3. The column number 2 implies that its traversal position along the “ne” curve at the current level is 1.

## 4 Curve Traversal and Indexing

Using the movement specification table or the specification tables described in Section 3, we can efficiently traverse along a space-filling curve or perform various indexing operations on any point along the curve. We describe three of them in this paper.

### 4.1 Curve Traversal

A key part of our table-driven approach is the *universal turtle traversal algorithm*, which traverses a self-similar space-filling curve based on a movement specification table. The algorithm is general enough for generating a variety of curves and efficient enough for use in applications that want to order computation along a specific curve. To traverse a curve of  $n$ -dimensions, it starts from the  $n$ -dimensional point  $\vec{0}$  with an integer extent along each dimension that the curve will cover. The initial value of the table index variable should point to a table row matching the primitive curve approximation of the curve. The algorithm first checks if it has reached the last recursion level by calling *IsBaseCase*. If true, it calls *BaseAction* to print the primitive curve or perform certain calculations by calling a routine that contains the calculations, otherwise, it divides the current space into subspaces and calls *Uturtle* for each subspace recursively. Then it calls a move routine to connect or move from an ending point of a curve in one subspace to the starting point of the curve in the next subspace.

Our algorithm is more general than previous algorithms in several aspects. First, the algorithm itself is curve- and dimensionality- independent. Second, each dimension may have a different extent. Third, the number of recursion levels can be controlled by adjusting the subspace size at the last recursion level. By controlling the number of recursion levels in different subspaces, we can also apply the algorithm to

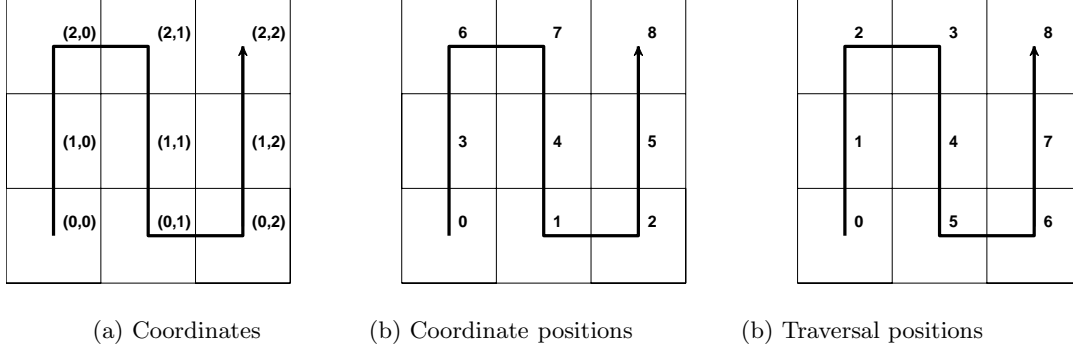


Figure 2: Indexing a 2D Peano curve.

Table 2: Coordinates to traversal index position specification table for a 2D Peano curve.

Current Level	Next Level								
sw	sw, 8	nw, 3	sw, 2	se, 7	ne, 4	se, 1	sw, 6	nw, 5	sw, 0
se	se, 2	ne, 3	se, 8	sw, 1	nw, 4	sw, 7	se, 0	ne, 5	se, 6
nw	nw, 6	sw, 5	nw, 0	ne, 7	se, 4	ne, 1	nw, 8	sw, 3	nw, 2
ne	ne, 0	se, 5	ne, 6	nw, 1	sw, 4	nw, 7	ne, 2	se, 3	ne, 8

*Uturtle*( $P, c, e, table, row, args$ )  
 $P$ : routine to be called at base level;  
 $c$ : coordinates of a point in a multi-dimensional space;  
 $e$ : extent at current level;  
 $table$ : movement specification table;  
 $row$ : a table row index in  $table$ ;  
 $args$ : arguments to be passed to  $P$ ;  
1. **if** (!*IsBaseCase*( $e$ ))  
2. *Resize*( $e$ );  
3. **for** each column  $col$  in a row  
4. *Uturtle*( $P, c, e, table, table[row][col].nextRow, args$ );  
5. *Move*( $c, e, table[row][col]$ );  
6. **else** *BaseAction*( $P, c, e, args$ );

Figure 3: Universal turtle algorithm.

applications that require non-uniform refinement. To improve efficiency of the algorithm, each move is described by a 2-tuple ( $dim, distance$ ) instead of a distance vector. For curves with moves only along coordinate axes (e.g. Hilbert and Peano curves),  $dim$  represents the axis of the move, and  $distance$  represents a movement distance, which can be a unit forward or backward along the axis. For curves with diagonal moves,  $dim$  specifies the axis with lowest traversal priority and  $distance$  specifies the distance along this dimension. A specialized implementation of the move function for such curves interprets this encoding. This axial movement specification eliminates unnecessary element-wise checking and calculation

for each move, particularly at the lowest recursion level. The ( $dim, distance$ ) representation naturally describes moves in many space-filling curves including the Hilbert and the Peano curves. It substantially reduces the low-level operations for other curves including the Morton and the Sierpiński curves.

## 4.2 Computing traversal position

Using the *c2i* table, we compute traversal position of a point along a space-filling curve from its coordinates in the space. Assume  $c, l, d$  are, respectively, the coordinates of a given point, the recursion level of the curve, and the dimensionality of the space.  $table$  and  $initrow$  are the position specification table and the initial table row that represents the primitive curve approximation of the curve. The algorithm first initializes the traversal position  $p$  to 0 and sets the current table row to the initial table row. It then concatenates the coordinates of the point  $c$  into variable  $coord = c_d c_{d-1} \dots c_1$ , where  $c_i = c_i^1 c_i^2 \dots c_i^l$  ( $1 \leq i \leq d$ ) represents the coordinate in dimension  $i$ .  $coord$  is then bit transposed into  $c^1 c^2 \dots c^l$ , where  $c^j = c_d^j c_{d-1}^j \dots c_1^j$  ( $1 \leq j \leq l$ ). For each recursion level  $l'$ , the algorithm extracts coordinate position at level  $l'$ , finds the table entry, gets the traversal position at level  $l'$  from the table entry, and updates the traversal position  $p$ . It updates the table row based on the current table entry before it checks the next level.

Table 3: Traversal index to coordinates position specification table for a 2D Peano curve.

Current Level	Next Level								
sw	sw, 8	se, 5	sw, 2	nw, 1	ne, 4	nw, 7	sw, 6	se, 3	sw, 0
se	se, 6	sw, 3	se, 0	ne, 1	nw, 4	ne, 7	se, 8	sw, 5	se, 2
nw	nw, 2	ne, 5	nw, 8	sw, 7	se, 4	sw, 1	nw, 0	ne, 3	nw, 6
ne	ne, 0	nw, 3	ne, 6	se, 7	sw, 4	se, 1	ne, 2	nw, 5	ne, 8

By using the precomputed position specification table, our curve indexing algorithm is faster than the previous byte-oriented algorithm. Our algorithm trades table space for execution speed. The size of the tables needed is modest. For example, our tables only contain 12, 32, 80, 192, and 448 table rows for Hilbert curves in three to seven dimensions, respectively.

### 4.3 Computing coordinates

Similarly, we can compute the coordinates of a point from its traversal position along a curve. The *i2c* algorithm uses an *i2c* position specification table. It extracts the traversal position of the point at each level, indexes the table entry in the *i2c* table, and retrieves the coordinate position at that level from the table entry. After it collects the coordinate positions from all levels, it performs a similar bit transpose to obtain coordinates of the point.

## 5 Reordering Computation Using SFCGen

SFCGen can be used for reordering data and computation in scientific applications through curve traversal and indexing. We only discuss computation reordering with the turtle traversal algorithm in this paper. Suppose we have a routine `prog` which contains the computation we want to apply to a sequence of subspaces ordered along a space-filling curve. Once the specification tables have been generated, to use SFCGen, we only need to replace `prog` with the following call to our traversal routine `turtle` and pass `prog` and its arguments `args` to `turtle`.

```
turtle(prog, origin, bounds, args);
```

`origin` represents the coordinates of the starting point. `bounds` is the bounds of an iteration space. When the turtle traversal algorithm reaches the base level where extents of one or more dimensions are less than or equal to a predefined minimum extent, it calls routine `BaseAction` with the coordinates of the current

```
do j = 1, n2
  do k = 1, n3
    do i = 1, n1
      c(i,j) = c(i,j) + a(i,k) * b(k,j)
```

Figure 4: Matrix multiplication.

point in the iteration space along with the routine `prog` and its arguments. `BaseAction` invokes `prog` only if the current point is within the iteration space. A sketch of the `BaseAction` routine is shown below.

```
void BaseAction(FP prog, C &c, va_list args) {
  int c1, c2, c3;
  c1 = c.GetDim(0);
  c2 = c.GetDim(1);
  c3 = c.GetDim(2);
  /* if coordinates are in bounds, call prog */
  if (c1 <= ub1 && c2 <= ub2 && c3 <= ub3) {
    (*prog)(c1, c2, c3, args);
  }
};
```

Consider using a space-filling curve to order a matrix multiplication computation shown in Figure 4. Let each point along the curve represent the partial computation of multiplying a pair of blocks from the input matrices. Using a space-filling curve to order this computation improves cache reuse of blocks over a conventional row or column based traversal orders of the blocks. At the base level, `prog` is called to perform computation in one subspace. Using matrix multiplication as an example, the computation in `prog` can be modified as follows.

```
do j = c2, min(c2+blk2-1,n2)
  do k = c3, min(c3+blk3-1,n3)
    do i = c1, min(c1+blk1-1,n1)
      c(i,j) = c(i,j) + a(i,k) * b(k,j)
```

`c1`, `c2`, and `c3` are the coordinates of the current point. `blk1`, `blk2`, and `blk3` are dimension extents of the `i`, `j`, and `k` dimensions of the subspace at the base level. Figure 5 shows computation reorderings in a 3D space  $2048 \times 1024 \times 1500$  using a Hilbert curve with two different minimum block sizes at the

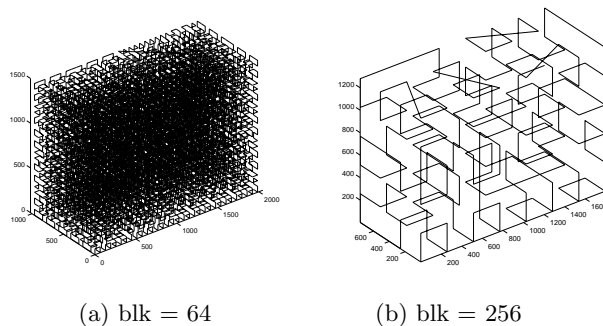


Figure 5: Reordering computation using a Hilbert curve in 3D space  $2048 \times 1024 \times 1500$ .

base level. Applying a similar reordering with block size of 16 or 64 to two different iteration spaces,  $2048 \times 1024 \times 1500$  and  $2048 \times 2048 \times 2048$ , we measured the execution performance of the matrix multiplication on a 300 MHz SGI Origin 2000 machine with a 32KB L1 cache and an 8MB L2 cache. We collected the number of cycles, L1 cache misses, L2 cache misses, and TLB misses. Table 4 shows a comparison of four different implementations of matrix multiply. All source codes were compiled using the MIPSpro cc/CC 7.3.1.3m compiler with either “-O2” or “-O3” as specified in Table 4. `orig` refers to matrix multiplication using the loop nest shown in Figure 4; `atlas` refers to the optimized implementation of matrix multiply generated by ATLAS 2.0 [27]; `blas` refers to the DGEMM routine from the BLAS library on the machine; `hil16` and `hil64` are the implementations of matrix multiply described earlier in this section, which use a 3D space-filling curve with block sizes of 16 and 64 to order computation.

Our measurements show that matrix multiply after computation reordering along a space-filling curve can be fast: when multiplying  $2048^2$  matrices, it was faster than both the `blas` and `atlas` implementations. For the test on non-square matrices, using the space-filling curve ordering was faster than `atlas`, but not quite as fast as the `blas`. The main point of this example is to show that space-filling curve traversal can be used to optimize performance by managing locality. However, if our space-filling curve traversal algorithm had not been efficient, any performance benefit due to improved locality could have been overshadowed by the cost of traversing the curve.

Table 4: Performance (counts in millions) of matrix multiplication.

Codes	Cycles	L1	L2	TLB
		misses	misses	misses
$a(2048, 2048) \times b(2048, 2048)$				
<code>orig-02</code>	82907	2286	471	2.29
<code>orig-03</code>	9988	1047	6.98	1.79
<code>blas</code>	19248	2478	5.36	0.22
<code>atlas-02</code>	56938	2160	526	291
<code>atlas-03</code>	10118	1048	7.20	1.80
<code>hil16-02</code>	29936	334	3.17	12.2
<code>hil64-03</code>	9862	970	3.09	12.6
$a(2048, 1500) \times b(1500, 1024)$				
<code>orig-02</code>	28903	838	161	0.84
<code>orig-03</code>	3613	218	2.05	0.56
<code>blas</code>	3466	280	1.33	0.07
<code>atlas-02</code>	19263	393	161	105
<code>atlas-03</code>	3680	219	2.20	0.56
<code>hil16-02</code>	10963	113	1.20	3.48
<code>hil64-03</code>	3599	225	1.20	1.30

## 6 Experimental Results

We evaluated the performance of our curve generation algorithm by comparing it to an efficient implementation of Butz’s byte-oriented algorithm [9] by Doug Moore at Rice University [22]. Our algorithm is implemented in C++, whereas Butz’s is implemented in C. We compiled all of the programs using the vendors’ compilers with “-O3”. We evaluated the performance of these algorithms on three modern microprocessor-based platforms: a 300MHz SGI Origin 2000, a 2.4GHz Intel Pentium 4 PC, and a 900MHz Intel Itanium 2 workstation. The compilers we used in the experiments are MIPSpro cc/CC 7.3.1.3m on the Origin 2000, Gnu C++ V2.96 on the Pentium 4, and Intel C++ V8.0 on the Itanium 2. For each execution of an algorithm, we measured its number of graduated instructions and execution time using hardware performance counters. The number of instructions and the number of cycles per visited point on a space-filling curve are presented. Each experiment was repeated multiple times. Variations between different executions were small and average measurements are presented. To access the hardware counters, we used `ssrun` on the Origin 2000 and PAPI [7] on the Pentium 4 and Itanium 2.

Table 5 and 6 show the average execution costs of visiting a point along 2D and 3D Hilbert curves at different levels using the two algorithms. Columns marked as “Cycles” and “GInst” represent the number of execution cycles and the number of graduated

Table 5: Performance comparison for traversing a 2D Hilbert curve.

Level	Butz & Moore						SFCGen					
	Origin 2000		Pentium 4		Itanium 2		Origin 2000		Pentium 4		Itanium 2	
	Cycles	GInst	Cycles	GInst	Cycles	GInst	Cycles	GInst	Cycles	GInst	Cycles	GInst
6	349	463	1479	996	366	676	39	39	39	39	14	34
8	387	541	1708	1210	418	794	28	40	40	40	13	32
10	436	630	2040	1441	476	922	27	40	39	40	13	32
12	485	712	2314	1655	529	1040	27	40	39	40	13	33
14	532	793	2589	1870	581	1158	27	40	38	40	13	32

Table 6: Performance comparison for traversing a 3D Hilbert curve.

Level	Butz & Moore						SFCGen					
	Origin 2000		Pentium 4		Itanium 2		Origin 2000		Pentium 4		Itanium 2	
	Cycles	GInst	Cycles	GInst	Cycles	GInst	Cycles	GInst	Cycles	GInst	Cycles	GInst
4	317	415	1323	883	317	617	43	39	36	34	12	24
5	338	476	1519	1028	357	698	28	39	32	35	17	43
6	373	526	1667	1159	392	770	26	38	32	35	10	24
7	402	578	1831	1286	425	842	26	38	32	34	17	43
8	426	617	1990	1415	455	914	25	38	32	34	10	24
9	458	676	2220	1564	494	998	25	38	31	35	17	43

instructions respectively. Compared to Butz’s non-recursive byte-oriented algorithm, our algorithm uses only 2% to 9% as many instructions on all platforms. Overall, our algorithm is faster than Moore’s efficient implementation of Butz’s byte-oriented algorithm by a factor of 9 to 68 on 2D cases and a factor of 7 to 72 on 3D cases. We are able to access a point along a Hilbert curve in 10 to 43 cycles while Butz’s algorithm needs hundreds to thousands cycles. This is mainly because our table-driven algorithm uses many fewer instructions with the precomputed information in the movement specification table. This also leads to more efficient execution with fewer cycles per graduated instruction on the Intel machines.

Table 7 shows comparisons of execution time for computing the traversal position of a point along a 2D Hilbert curve from its coordinate position in the columns under “*c2v*” and execution time for computing the coordinates of a point from its traversal position along the curve in columns under “*i2c*”. The top part of the table show the results from Moore’s implementation of Butz’s algorithm. Results in the bottom part of the table are from our indexing algorithm. The results show that our algorithm performs from 70% to a factor of 11 faster than the byte-oriented algorithm across all three platforms. Our table-based indexing leads to 60% to about a factor of 5 less graduated instructions and more efficient execution with

the precomputed position information. The less significant improvement observed on the Intel Pentium 4 is caused by the relatively higher cost of bit transpose that both algorithms use to retrieve coordinate information at each level from the original coordinates or to collect the coordinate information from each level. Overall, we see more improvement in the “*c2v*” indexing. The results in Table 5 and Table 7 also tell us that indexing a point along a space-filling curve is much more expensive than traversing points on the curve.

We also measured the performance of traversing and indexing a Morton curve and a Peano curve on the Origin 2000 and Itanium 2 using our table-based algorithms. Table 8 shows the average number of cycles for visiting a point in curve traversal or for indexing a point along a 2D Morton or Peano curve in the columns under “*traversal*”, “*c2v*” or “*i2c*” respectively. The recursion levels at which we collected results are lower for the Peano curve since it has higher refinement factor 3. However the results are fairly consistent across the levels. The results show that traversing a Morton curve is more expensive than traversing the other two space-filling curves because the diagonal moves in a Morton curve cause more calculation. On the other hand, indexing a Peano curve is less efficient mainly because expensive division operations are needed with the refinement factor of 3.

Table 7: Performance comparison for indexing a 2D Hilbert curve.

Level	<i>Origin 2000</i>				<i>Pentium 4</i>				<i>Itanium 2</i>			
	<i>c2i</i>		<i>i2c</i>		<i>c2i</i>		<i>i2c</i>		<i>c2i</i>		<i>i2c</i>	
	<i>Cycles</i>	<i>GInst</i>	<i>Cycles</i>	<i>GInst</i>	<i>Cycles</i>	<i>GInst</i>	<i>Cycles</i>	<i>GInst</i>	<i>Cycles</i>	<i>GInst</i>	<i>Cycles</i>	<i>GInst</i>
<i>Butz &amp; Moore</i>												
6	1236	1040	1068	928	1684	993	1479	996	604	891	388	689
8	1506	1258	1272	1137	1968	1199	1724	1210	765	1092	441	816
10	1680	1473	1477	1366	2120	1377	2048	1441	859	1234	502	953
12	1833	1625	1685	1582	2247	1478	2312	1657	948	1349	557	1080
14	2018	1812	2038	1775	2510	1639	2592	1870	1039	1459	601	1170
<i>SFCGen</i>												
6	134	240	146	272	864	607	795	603	136	257	144	321
8	133	268	157	317	1124	749	1005	746	156	296	159	387
10	158	309	180	361	1224	840	1157	884	165	343	174	453
12	170	333	205	408	1293	882	1347	1025	171	373	189	519
14	190	375	225	453	1404	972	1527	1162	179	423	203	579

Table 8: Performance of traversal and indexing with 2D Morton and Peano curves.

Level	<i>Morton</i>						Level	<i>Peano</i>					
	<i>Origin 2000</i>			<i>Itanium 2</i>				<i>Origin 2000</i>			<i>Itanium 2</i>		
	<i>traversal</i>	<i>c2i</i>	<i>i2c</i>	<i>traversal</i>	<i>c2i</i>	<i>i2c</i>		<i>traversal</i>	<i>c2i</i>	<i>i2c</i>	<i>traversal</i>	<i>c2i</i>	<i>i2c</i>
6	56	149	137	46	156	134	4	35	338	600	17	492	774
8	49	143	148	46	177	148	5	29	400	731	16	598	966
10	48	171	175	46	184	170	6	28	470	870	16	706	1160
12	48	168	184	46	188	174	7	28	540	1008	16	810	1351
14	48	201	212	46	194	187	8	28	610	1147	16	916	1544

Traversing and indexing Hilbert curves are both very efficient.

## 7 Conclusions

Space-filling curves have been used in many fields for transforming multi-dimensional problems into one-dimensional forms. In scientific computing, using space-filling curves for ordering data or computation can improve data locality. When applying these curves in practice, the cost of curve traversal or indexing is important as it can contribute significantly to overall execution time. For this reason, efficient indexing and traversal algorithms are important. This paper describes a table-based strategy for efficiently enumerating coordinates or indexing points along space-filling curves in two or more dimensions. The higher performance of our traversal and indexing methods make them practical to apply at a finer grain such as for ordering a matrix multiplication computation.

## Acknowledgements

This research was supported in part by NCSA under National Science Foundation cooperative agreement ACI-9619019 and the Los Alamos National Laboratory Computer Science Institute (LACSI) through LANL contract number 03891-99-23 as part of the prime contract (W-7405-ENG-36) between the DOE and the Regents of the University of California.

## References

- [1] D. Abel and D. Mark. A comparative analysis of some two-dimensional orderings. *International J. of Geographical Information and Systems*, 4(1):21–31, 1990.
- [2] C. J. Alpert and A. B. Kahng. Multi-way partitioning via spacefilling curves and dynamic programming. In *Proceedings of the 31st Annual Conference on Design Automation Conference*, pages 652–657, 1994.

- [3] J. Bartholdi III and P. Goldsman. Vertex-labeling algorithms for the Hilbert spacefilling curve. *Software-Practice and Experience*, 31:395–408, 2001.
- [4] T. Bially. Space-filling curves: Their generation and their application to bandwidth reduction. *IEEE Trans. on Information Theory*, IT-15(6):658–664, Nov. 1969.
- [5] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33:322–373, 2001.
- [6] G. Breinholt and C. Schierz. Algorithm 781: Generating Hilbert’s space-filling curve by recursion. *ACM Trans. on Mathematical Software*, 24(2):184–189, 1998.
- [7] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Dallas, TX, Nov 2000.
- [8] A. R. Butz. Space filling curves and mathematical programming. *Information and Control*, 12:314–330, 1968.
- [9] A. R. Butz. Alternative algorithm for Hilbert’s space-filling curve. *IEEE Trans. on Computers*, C-20(4):424–426, Apr. 1971.
- [10] M. Challacombe. A general parallel sparse-blocked matrix multiply for linear scaling scf theory. *Computer Physics Communications*, 128(1-2):93–107, 2000.
- [11] A. J. Cole. A note on space filling curves. *Software-Practice and Experience*, 13(12):1181–1189, Dec. 1981.
- [12] A. J. Fisher. A new algorithm for generating hilbert curves. *Software-Practice and Experience*, 16(1):5–12, 1986.
- [13] L. M. Goldschlager. Short algorithms for space-filling curves. *Software-Practice and Experience*, 11(1):99–100, Jan. 1981.
- [14] Y. C. Hu, A. Cox, and W. Zwaenepoel. Improving fine-grained irregular shared-memory benchmarks by data reordering. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Dallas, TX, 2000.
- [15] G. Jin and J. Mellor-Crummey. SFCGen: A framework for efficient generation of multi-dimensional space-filling curves by recursion. *ACM Trans. on Mathematical Software*, 31(1), 2005.
- [16] G. Jin, J. Mellor-Crummey, and R. Fowler. Increasing temporal locality with skewing and recursive blocking. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, Denver, CO, Nov 2001.
- [17] X. Liu and G. Schrack. Encoding and decoding the hilbert order. *Software-Practice and Experience*, 26:1335–1346, 1996.
- [18] X. Liu and G. Schrack. An algorithm for encoding and decoding the 3-d hilbert order. *IEEE Transactions on Image Processing*, 6:1333–1337, 1997.
- [19] Y. Matias and A. Shamir. A video scrambling technique based on space filling curve. In *Advances in Cryptology – CRYPTO’89*, pages 398–416, 1987.
- [20] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 13th ACM International Conference on Supercomputing*, pages 425–433, Rhodes, Greece, June 1999.
- [21] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming*, 29(3):217–247, June 2001. Special issue with selected papers from 13th ACM International Conference on Supercomputing (Rhodes, Greece, June 1999).
- [22] D. Moore. <http://www.caam.rice.edu/~dougm/twiddle/Hilbert>. Feb. 2000.
- [23] L. K. Platzman and J. J. Bartholdi, III. Spacefilling curves and the planar travelling salesman problem. *J. of the ACM*, 36:719–737, 1989.
- [24] P. Prusinkiewicz, A. Lindenmayer, and F. D. Fracchia. Synthesis of space-filling curves on the square grid. In H. O. Peitigen, J. M. Henriques, and L. F. Pendo, editors, *Fractals in the Fundamental and Applied Sciences*, pages 341–366. Elsevier Science Publisher BV, Amsterdam, The Netherlands, 1991.
- [25] H. Sagan. *Space-Filling Curves*. Springer-Verlag, New York, NY, 1994.
- [26] J. Salmon and M. Warren. Parallel out-of-core methods for n-body simulation. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.
- [27] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27:3–25, 2001.
- [28] I. H. Witten and B. Wyvill. On the generation and use of space-filling curves. *Software-Practice and Experience*, 13:519–525, 1983.
- [29] Y. Zhang and R. E. Webber. Space diffusion: An improved parallel halftoning technique using space-filling curves. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, pages 305–312, 1993.