Syracuse University

SURFACE

Electrical Engineering and Computer Science College of Engineering and Computer Science

1997

Parallel Domain Decomposition and Load Balancing Using Space-Filling Curves

Srinivas Aluru
Syracuse University

Faith E. Sevilgen
Syracuse University, School of EECS, sevilgen@top.cis.syr.edu

Follow this and additional works at: https://surface.syr.edu/eecs

Part of the Computer Sciences Commons

Recommended Citation
Aluru, Srinivas and Sevilgen, Faith E., "Parallel Domain Decomposition and Load Balancing Using Space-Filling Curves" (1997). *Electrical Engineering and Computer Science*. 34.
https://surface.syr.edu/eecs/34

This Working Paper is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Electrical Engineering and Computer Science by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

Parallel Domain Decomposition and Load Balancing Using Space-Filling Curves*

Srinivas Aluru
Dept. of CS
New Mexico State University
Las Cruces, NM 88003-8001
email: *aluru@cs.nmsu.edu*

Fatih E. Sevilgen
School of EECS
Syracuse University
Syracuse, NY 13244-4100
sevilgen@top.cis.syr.edu

Abstract

Partitioning techniques based on space-filling curves have received much recent attention due to their low running time and good load balance characteristics. The basic idea underlying these methods is to order the multidimensional data according to a space-filling curve and partition the resulting one-dimensional order. However, space-filling curves are defined for points that lie on a uniform grid of a particular resolution. It is typically assumed that the coordinates of the points are representable using a fixed number of bits, and the run-times of the algorithms depend upon the number of bits used.

In this paper, we present a simple and efficient technique for ordering arbitrary and dynamic multidimensional data using space-filling curves and its application to parallel domain decomposition and load balancing. Our technique is based on a comparison routine that determines the relative position of two points in the order induced by a space-filling curve. The comparison routine could then be used in conjunction with any parallel sorting algorithm to effect parallel domain decomposition.

1 Introduction

Many scientific and engineering applications involving iterative methods can be represented by computational graphs [11]. The nodes of the computational graph represent tasks that can be executed concurrently. The edges of the graph represent the communication required between tasks from one iteration to the next. Computational graphs derived from many applications are such that the nodes correspond to two- or three-dimensional coordinates and the edges are limited to vertices that are physically proximate. Examples of such applications include finite element methods, PDE solvers and molecular dynamics simulations. To parallelize the application on a collection of p processors, the computational graph should be partitioned to the p processors. The partitioning problem has three requirements: balancing workload, minimizing interprocessor communication and minimizing the running time of the partitioning algorithm itself. It is a very difficult task to fulfill these conflicting requirements.

There is currently a significant interest in the research community for using space-filling curves for partitioning multidimensional graphs. Ou [12] and Pilkington et al. [14] have shown that space-filling curve based partitioning techniques provide considerably good quality partitions with very low costs. Similar techniques have been used by Parashar et al. [13] and Warren et al. [16] for parallel Adaptive Mesh Refinement and the N-body problem, respectively. The underlying idea is to map multidimensional data to one dimension where the partitioning is trivial. There are many ways to map multidimensional data to one dimension. However, a mapping that could be used in partitioning algorithms should preserve the proximity information present in the multidimensional space to minimize communication costs. Experimental [1, 6] and analytical [9, 7, 3] studies show that space-filling curves provide such mappings. Unfortunately, space-filling curves are defined for points that lie on a uniform grid of a particular resolution [15]. It is typically assumed that the coordinates of the points are representable using a fixed number of bits, and the run-times of the algorithms depend upon the number of bits used. Let l be the largest distance and s be the smallest distance between any pair of points. If all the points are known in advance, l and s can be computed beforehand and $\lceil \log \frac{l}{s} \rceil$ bits are sufficient to represent the points. Thus, current algorithms have a running time proportional to $\log \frac{l}{s}$ and are difficult to use for arbitrary and dynamic points.

In this paper, we present a simple technique for ordering arbitrary and dynamic multidimensional data using space-filling curves. For the Z-curve and the Graycode space-filling curves, we present comparison routines that find which of given two points appears first in the order induced by the space-filling curve. The comparison routines takes $O(d \log\log \frac{l}{s})$ time for

*This research is supported by the Sandia National Laboratories and National Science Foundation under CCR-9702991.

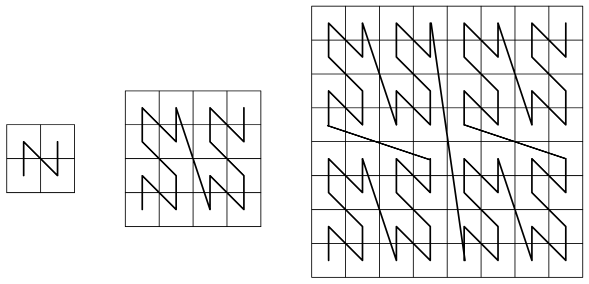

Figure 1: Z-curve for grids of size 2×2, 4×4 and 8×8.

two d-dimensional points, where l and s need not be known in advance. Moreover, if bit shifting is assumed to be a constant time operation, the comparison routines take only $O(d)$ time. Our method not only makes the application of space-filling curves to arbitrary points more efficient but also makes it possible to apply these strategies in the dynamic case.

2 Space-Filling Curves

In this section, we describe the notion of space-filling curves and describe some of the popularly used curves. We first restrict our attention to square two-dimensional grids of size $2^k \times 2^k$. Space-filling curves can be described in a variety of ways. They can be described recursively. The curve for a $2^k \times 2^k$ grid is composed of four $2^{k-1} \times 2^{k-1}$ grid curves. The curves can also be specified by bit interleaving - the position of a grid point along the curve can be described by interleaving the bits of the coordinates of the point with the interleaving function being a characteristic of the curve.

Z-Curve. The Z-curves [10] for 2×2, 4×4 and 8×8 grids are shown in Figure 1. The curve for a $2^k \times 2^k$ grid is composed of four $2^{k-1} \times 2^{k-1}$ grid curves one in each quadrant of the $2^k \times 2^k$ grid. The order in which the curves are connected is the same as the order of traversal of the 2×2 curve, and this holds true for all space-filling curves. Alternatively, given a point in two dimensions and assuming that each coordinate is represented using k bits, the position of the point on the Z-curve can be obtained by interleaving the bits representing the x and y coordinates, starting from the x coordinate. For example, $(3, 5) = (011, 101)$ translates to $011011 = 27$. All the space-filling curves can also be described in such a fashion using a suitable bit-interleaving function.

Graycode Curve. The graycode curve [4] uses the same bit interleaving function as the Z-curve. While the Z-curve visits points in the numerical order of the results of applying the interleaving function to the

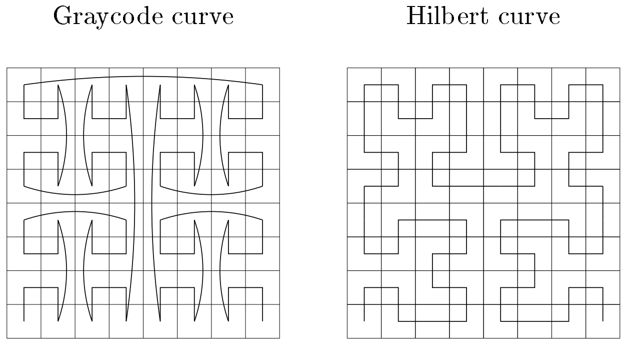

Figure 2: Graycode and Hilbert curves for grids of size 8×8.

points, the graycode curve visits them in the graycode order (see Figure 2). The curve can be described recursively as follows: Place four curves of the previous resolution in the four quadrants. Flip the upper quadrant curves once around one of the coordinate axes and then around the other axis. The curves in the lower quadrants are left as they are.

Hilbert Curve. The Hilbert curve [5] is a smooth curve that avoids the sudden jumps present in the Z and the Graycode curves (Figure 2). The curve is composed of four curves of the previous resolution placed in the four quadrants. The curve in the lower left quadrant is rotated clockwise by 90^o and lower right one is rotated counterclockwise by 90^o.

The curves can be easily extended for cases when the dimension spans are uneven and for higher dimensional grids. Consider a $2^k \times 2^m$ grid $(k > m)$. One can use 2^{k-m} curves each for a $2^m \times 2^m$ grid and place them next to each other. If needed, the curves can be tilted upside down as appropriate, to avoid sudden jumps. A higher dimensional curve can be obtained by using a template of the same dimension (such as a $2 \times 2 \times 2$ grid for three dimensions). A detailed description of space-filling curves for uneven dimension spans and higher dimensions can be found in [6].

3 Space-Filling Curves for Arbitrary Points

Consider the problem of ordering a given set of n points in the order in which a particular space-filling curve visits them. We will use square grids in two-dimensions for ease of presentation. The same techniques can be easily extended to higher dimensions.

The traditional method relies on the assumption that the coordinates of the points are representable using k-bit integers (for some fixed k). An index is calculated by interleaving the bits of these integers with the interleaving function of the space-filling curve. Sorting these indices gives the order of the points according to the space-filling curve. Although the method is very simple and easy to implement it does not solve the problem for arbitrary points. If $\frac{l}{s}$, the ratio of the largest distance to the smallest distance between any two points, is greater than 2^k, the coordinate values of the points cannot be represented by k-bit integers distinctly.

For arbitrary points, we could adopt the following strategy: We start with a square (*root cell*) large enough to contain all of the given n points. We can continuously subdivide the square into grids of resolution $2^k \times 2^k$ (starting with $k = 0$ and increasing k) until no grid cell contains more than one point. As each grid cell contains only one point, the order in which the curve visits the grid cells also determines the order in which the curve visits the points. The running time of this algorithm is proportional to k, which depends on the distribution of the points. In the worst case, $\frac{l}{s}$ grid lines are necessary to separate the closest points. Thus, the upper bound on k is $O(\log \frac{l}{s})$.

Before we proceed to describe an algorithm with a better running time, it is necessary to introduce some terminology: Assume a sequence of $2^k \times 2^k$ grids ($0 \leq k < \infty$) imposed on the root cell. We use the term *cell* to describe each grid cell of any of these hypothetical grids. A cell can be subdivided into four subcells of equal size. Let us name the subcells with respect to the cell as though the center of the cell is the origin and the horizontal and vertical lines denote the x and y axes, respectively. We can label the upper right subcell as subcell I (following the usual quadrant I notation) and so on. The root cell contains 2^{2k} cells of length $\frac{l}{2^k}$. A line is called a *k-boundary* if it contains an edge of a cell of length $\frac{l}{2^k}$.

The procedure we adopt to order given n points according to a given space-filling curve is as follows: We will devise a comparison routine that takes two points as input and determines which of the two is first visited by the space-filling curve. This is done by first determining the smallest subcell of the root cell that encloses the two points followed by determining the order in which the curve visits the four subcells of this subcell. Note that the order in which the space-filling curve visits two points is independent of the position and the presence of other points. Once we design such a comparison routine, we can use any sorting algorithm to order the points. In the following, we describe such a technique for some space-filling curves.

Consider a rectangle R that has the given two points at the opposite ends of a diagonal. The smallest subcell of the root cell enclosing the two points is also the smallest subcell that encloses the rectangle R, which we will determine. We use the technique proposed by Aluru et al. [2] for this purpose. The smallest subcell of the root cell enclosing R is of size $\frac{l}{2^{k-1}}$, where k is the smallest number such that a *k-boundary* crosses the rectangle R (see figure 3). To determine this, we examine boundaries parallel to each coordinate axis in turn.

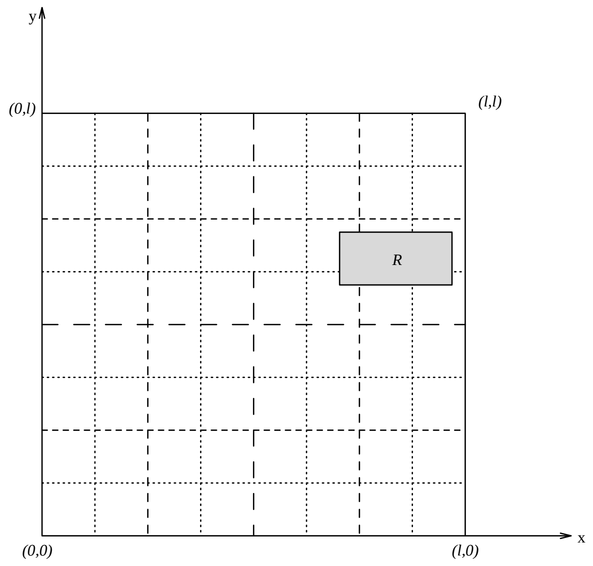

Figure 3: A *root cell* of length l and a rectangle R. The big dashed lines are 1-*boundaries*, the small dashed lines are 2-*boundaries* and the dotted lines are 3-*boundaries*. 2-*boundaries* are also 3-*boundaries* and 1-*boundaries* are also 2-*boundaries* and 3-*boundaries*.

Consider boundaries parallel to the y-axis. These can be specified by their distance from the y-axis. The family of *k-boundaries* is specified by $i\frac{l}{2^k}$, $0 \leq i \leq 2^k$. We need to find the smallest integer k such that a *k-boundary* parallel to y-axis passes through R, i.e. the smallest k such that $x_{min} < i\frac{l}{2^k} < x_{max}$ for some i. By minimality of k, only one *k-boundary* passes through R. Let j be the smallest integer such that $\frac{l}{2^j} < (x_{max} - x_{min})$. $j = \lceil \log_2 \frac{l}{x_{max} - x_{min}} \rceil$. There is at least 1 and at most 2 *j-boundaries* passing through R. These boundaries are given by $h_1 = \lceil \frac{2^j x_{min}}{l} \rceil \frac{l}{2^j}$ and $h_2 = \lfloor \frac{2^j x_{max}}{l} \rfloor \frac{l}{2^j}$. Since $k \leq j$, any *k-boundary* is also a *j-boundary*, forcing the *k-boundary* passing through R to coincide with h_1 or h_2. Let a be $\lceil \frac{2^j x_{min}}{l} \rceil$. $h_1 = a\frac{l}{2^j}$ and $h_2 = h_1$ or $(a+1)\frac{l}{2^j}$. If $h_2 \neq h_1$, let a' be the even integer among a and $a+1$. Otherwise, let a' be equal to a. It is clear that $j - k$ is equal to the highest power of 2 that divides a'. One way to find this is $j - k = \log_2(1 + \{a' \oplus (a' - 1)\}) - 1$.

We use the computation of the smallest cell enclosing a given rectangle as a basic operation. This operation requires the computation of logarithm and exponent with base 2. If we assume that each coordinate of any point is represented using a single machine word, we do not need arbitrarily large exponents. Exponentiation to the base 2 can be accomplished by bit-shifting. We feel that bit shifting and computing the logarithm can be taken to be constant time operations, and experimental results indicate that it is reasonable to do so. Under these assumptions, the smallest cell can be found in $O(d)$ time for d dimensions.

Nevertheless, we are interested in a careful analysis of the complexity. Let l be the largest distance and s be the smallest distance between any pair of points. In the algorithm for computing the smallest cell enclosing two points, we first compute the j-boundary following which 2^j is computed. We need at most $\frac{l}{s}$ boundaries parallel to an axis to separate the closest points. Therefore, 2^j is at most $\frac{l}{s}$. We can compute 2^j in $O(\log j) = O(\log\log \frac{l}{s})$ time. This is an improvement over the $O(\log \frac{l}{s})$ factor present in other algorithms. However, we found that using bit shifting to compute 2^j is much faster in practice.

It is important to reflect more on the complexity of the basic operation of computing the smallest cell enclosing two points. An analogy can be drawn to sorting algorithms. An optimal sorting algorithm takes $O(n \log n)$ time assuming that two numbers can be compared in constant time. This is clearly not valid if we use multi-precision numbers or if we look at the bit complexity of comparing two numbers. We simply make the assumption that certain basic operations on machine words can be performed in constant time, which is reasonable in practice. In a similar manner, if bit shifting operation on a machine word is taken to be a constant time operation, computing the smallest cell requires only $O(d)$ time for d dimensions.

Once we find the smallest subcell, the order in which the space-filling curve visits the two points can be determined if we know the order in which the four subcells of the smallest subcell containing the two points is visited by the curve. For the Z-curve, this order is always subcell III, subcell II, subcell IV and subcell I, irrespective of the position and the size of the subcell. For example, if one of the points is in subcell I and the other is in subcell III, the point in subcell III is visited before the point in subcell I.

The order in which the graycode curve visits the subcells depends on the location of the cell in the grid it belongs to. Fortunately, there are only two different orders possible: subcell III, subcell II, subcell I, and subcell IV or subcell I, subcell IV, subcell III, and subcell II. We refer to the former as $\cap$-order and the latter as $\cup$-order. Each row of cells in a grid has a unique order. If we number the rows in a grid bottom-up, the $\cap$-order can only be found at the odd numbered rows and even numbered rows have $\cup$-order. To compare two points in graycode order, we check the location of the smallest cell. If it is on one of the odd numbered rows, the comparison is performed using the $\cap$-order and $\cup$-order is employed otherwise.

A similar comparison routine for the Hilbert curve seems to be more complicated and may be impossible in constant time because of the combined effect of two distinct rotations. In other words, all the bits of the coordinates of subcells are necessary to find the order for the Hilbert curve.

The comparison routines described above can be easily extended to d-dimensional space. In this case, the cells are d-dimensional hypercubes and the k-boundaries are $(d-1)$-dimensional hyperplanes perpendicular to an axis. For the smallest cell containing two points p_1 and p_2, we need to check each axis in turn and find the smallest integer k such that a k-boundary intersects the hyperrectangle R defined by p_1 and p_2. This procedure requires $O(d)$ operations. Subcells of the smallest cell containing the points and the order of the subcells can be determined in $O(d)$ time. Thus, comparison of two points requires $O(d)$ operations and has a bit-complexity of $O(d \log\log \frac{l}{s})$.

4 Parallel Domain Decomposition

To effect parallel domain decomposition on p processors, we adopt the following strategy: We first sort the data along the space-filling curve using a standard sorting algorithm. The resulting sorted data is cut into p equal parts and distributed to the p processors. Any parallel sorting algorithm can be used in conjunction with the comparison routines designed in the previous section. Parallel sorting also has the desirable side effect of distributing the sorted data. Hence, a separate distribution phase is not necessary.

We implemented the space-filling curve based partitioning algorithm described above in C with calls to the MPI on an Intel Paragon. We have employed parallel sample sort algorithm [8] because of its practical efficiency and scalability and chose the Z-curve. We tested our implementation with three different graph sizes and two types of distributions for each size: uniform and non-uniform. The uniform distributions are generated using a uniform random number generator. The graphs chosen for the non-uniform distributions

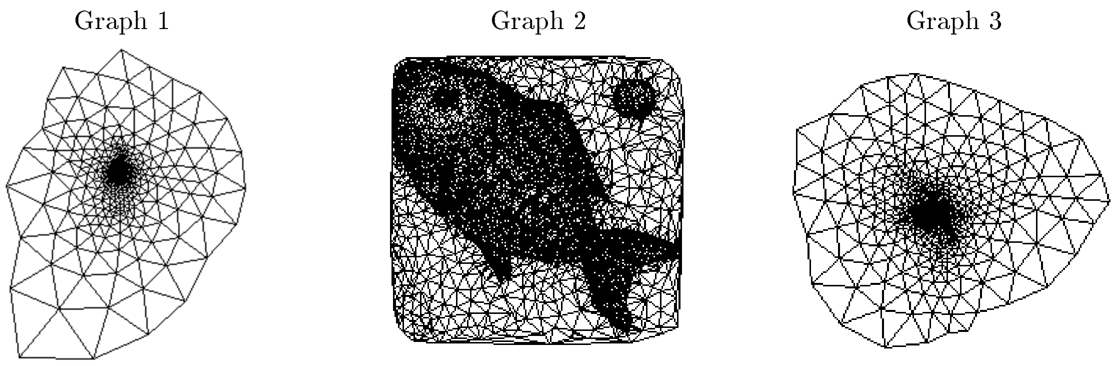

Figure 4: The graphs on which our results for non-uniform distributions are presented. Graph 1 has 6019 nodes, Graph 2 has 10166 nodes and Graph 3 has 15606 nodes.

are shown in Figure 4. The run times of parallel domain decomposition for different numbers of processors are shown in Table 1. The times are in seconds and are averaged over 10 runs.

The main purpose of this experiment is to check the running times for same graph sizes but different distributions. Apart from minor fluctuations attributable to the random nature of sample sort, the running times do not considerably differ from uniform distribution to non-uniform distribution. Note that there is a superlinear speedup for small processor sizes. This anomaly is because sample sort algorithm does not perform well for very small number of processors.

It is very important to be able to determine the processor a given data point is allocated to. The comparison routines described in the previous section can be used to solve this problem as well. We can use an array of length p to store the coordinates of the first point stored on each processor i.e. processor boundaries. By a simple binary search on this array using the prescribed comparison algorithm, the processor containing a given data point can be determined.

5 Parallel Incremental Load Balancing

Many scientific simulations requires the computational graph to change with time. Although these changes are very small compared to the whole graph, they may cause imbalance in the workload. Since load imbalance is one of the reasons of degrading performance of applications, the graph should be redistributed. However, it is very expensive to redistribute the graph from scratch each time when there is a change. There are two popular approaches to solve this problem: In the first approach, we wait until the changes exceed a threshold value following which load distribution is performed from scratch. In the second approach, we use the previous domain decomposition information to improve the performance of the load balancing algorithm, i.e. we perform incremental load balancing.

Space-filling curve based partitioning has two phases: sorting the multidimensional data and partitioning the ordered list to processors. After the changes in the graph, it may not remain sorted. If the changes are local to the processor, readjusting the graph locally in each processor is sufficient. However, there may be vertices that go beyond the processor boundaries. The idea is to determine the processors responsible for each of them and send them to the appropriate processors so that local readjustments are enough. We sort all the vertices that need to be sent to other processors and find the positions of the processor boundaries on this sorted array using binary search. For the sort and search operations, we employ the proposed comparison routines. After the processors responsible for the vertices are determined, an all-to-all communication is sufficient to exchange them. At this point, all the vertices within the processors are local. We sort the the vertices locally within the processors to achieve the global order. We now can partition it again to balance the workload. Since space-filling curves preserve locality, the changes are expected to be mostly local.

6 Conclusions and Open Problems

Partitioning workloads evenly among processors is an important problem in parallel and distributed computing. Such a problem is especially hard for non-uniform data in a multi-dimensional space. Space-filling curve based partitioning method is one of the widely used techniques because of its proximity preserving characteristic. The heart of this technique is

Number of Nodes	Graph size (# of vertices)					
	6019		10166		15606	
	non-uniform	uniform	non-uniform	uniform	non-uniform	uniform
2	19.32	29.28	75.25	102.60	87.44	176.71
4	6.33	6.53	11.27	10.12	19.09	15.66
8	2.44	2.19	4.61	3.92	7.79	6.54
16	1.20	1.27	2.29	2.30	3.50	3.47
32	1.91	1.74	2.08	2.43	2.78	3.43

Table 1: Parallel partitioning time (in seconds) for uniform and non-uniform graphs of different sizes.

to sort the multidimensional data according to a space filling curve. In this paper, we presented practically efficient algorithms to linearly order points in a multidimensional space using the Z-curve and the Graycode curve, without making any assumptions on the precision or distribution of the points. We have demonstrated the practicality of our technique using the Intel Paragon. Another popular curve used in partitioning algorithms is the Hilbert space-filling curve. It would be of much interest to determine if our technique can be extended to the Hilbert space-filling curve as well.

Space-filling curves are used for partitioning because they are "proximity preserving" mappings of multidimensional space to a one-dimensional space. However, the quality of partitioning obtainable by using space-filling curves has only been studied experimentally. Analytical studies are usually limited to cases where points occupy every cell in a fixed grid. It is computationally more efficient, especially in parallel, to use space-filling curves for ordering multidimensional data. Therefore, it is important to study the quality of the orderings obtainable by space-filling curves.

Acknowledgments

We wish to thank Chao Wei-Ou for providing us the computational graphs on which our experimental results about graph partitioning are presented. We are grateful to Sandia National Laboratories for allowing us to use their Paragon, on which our experimental results are reported.

References

[1] D.J. Abel, D.M. Mark, A comparative analysis of some two-dimensional orderings; *International Journal of Geographical Information Systems, 4(1)* (1990) 21-31.

[2] S. Aluru, G.M. Prabhu and J. Gustafson, Truly distribution-independent hierarchical algorithms for the N-body problem, *Proc. Supercomputing '94*, (1994) 420-428.

[3] T. Asano, D. Ranjan, T. Ross, E. Welzl and P. Widmayer, Spacefilling curves and their use in the design of geometric data structures, *Lecture Notes in Computer Science, 911* 36-48.

[4] C. Faloutsos, Gray codes for partial match and range queries, *IEEE Transactions on Software Engineering, 14(10)* (1988) 1381-1393.

[5] D. Hilbert, Uber die stegie Abbildung einer Linie auf Flachenstuck, *Math. Ann., 38* (1891) 459-460.

[6] H.V. Jagadish, Linear clustering of objects with multiple attributes, *Proc. ACM SIGMOD International Conference on the Management of Data*, (1990) 332-342.

[7] H.V. Jagadish, Analysis of the Hilbert curve for representing two-dimensional space, *Information Processing Letters, 62* (1997) 17-22.

[8] V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing,* The Benjamin/Cummings Publishing Co., 1994.

[9] B. Moon, H.V. Jagadish, C. Faloutsos and J.H. Saltz, Analysis of clustering properties of Hilbert space-filling curve, Technical Report No. CS-TR-3590, University of Maryland Department of Computer Science, March 1996.

[10] G.M. Morton, A computer oriented geodetic data base and a new technique in file sequencing, IBM, Ottawa, Canada (1966).

[11] C.W. Ou, *Partitioning and Incremental Partitioning of Adaptive Irregular Problems,* Ph.D. Thesis, Syracuse University, 1996.

[12] C.W. Ou and S. Ranka. Parallel Remapping Algorithms for Adaptive Problems, *Proc. Frontiers' 95,* (1995) 367-374.

[13] M. Parashar and J.C. Browne, Distributed Dynamic Data-Structures for Parallel Adaptive Mesh-Refinement, *Proceedings of the International Conference on High Performance Computing*, (1995).

[14] J.R. Pilkington and S.B. Baden, Dynamic partitioning of non-uniform structured workloads with space-filling curves, *IEEE Transaction on Parallel and Distributed Systems, 7(3)* (1996) 288-300.

[15] H. Sagan, *Space-filling curves*, Springer-Verlag, 1994.

[16] M.S. Warren and J.K. Salmon, A Parallel Hashed Oct-Tree N-Body Algorithm, *Proc. Supercomputing '93*, (1993) 12-21.