THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCE

MULTI-GPU SOLUTIONS OF GEOPHYSICAL PDES WITH RADIAL BASIS FUNCTION-GENERATED FINITE DIFFERENCES

By

EVAN F. BOLLIG

A Dissertation submitted to the
Department of Scientific Computing
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Fall Semester, 2012

Evan F. Bollig defended this dissertation on October 1, 2012.

The members of the supervisory committee were:

Gordon Erlebacher
Professor Directing Thesis

Natasha Flyer
Outside University Representative

Mark Sussman
University Representative

Dennis Slice
Committee Member

Ming Ye
Committee Member

Janet Peterson
Committe Member

The Graduate School has verified and approved the above-named committee members, and certifies that the dissertation has been approved in accordance with the university requirements.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

AN ALTERNATIVE STENCIL GENERATION ALGORITHM FOR RBF-FD

Like all RBF methods, RBF-FD is designed to handle irregular node distributions, so the emphasis in literature focuses on how the method manages point clouds. While nothing prevents implementations of RBF-FD from utilizing existing meshes/lattices, most work in the field concentrates on simple geometries to better understand properties of the method and develop extensions. Without mesh/lattice connectivity available, stencils are generated by choosing the n-nearest neighbors to a center node, inclusive of the center. This is known more formally as a *k-nearest neighbor (k-NN)* problem [22] (a.k.a. ℓ-nearest neighbor search [25]). Here "nearest" is defined with the Euclidean distance metric, although it is possible to generalize to other metrics (see e.g., [2]).

In comparison to the RBF-FD method, global RBF methods with infinite support connect all nodes to all other nodes, so there is no need for neighbor queries. On the other hand, compact RBF methods require all nodes—with no limit on the count—that lie within the support/radius of the RBF centered at each node. This type of neighbor query is referred to as a *ball query* (a.k.a. range query [25]) due to the closed ball created by the radius of support for a compact RBF (see Equation **??**).

The k-NN and ball query share many similarities, but the former can be harder to solve. Consider, for example, the scenario in Figure 1.1. Two ball queries around a green stencil center are represented as dashed and dash-dot circles. The inner query returns four neighbors, and the outer returns six. If a stencil of size $n = 6$ is desired, then the outer query can be truncated to give the five required neighbors shown in blue. In this example the red node and the farthest blue node are equidistant from the center, and ties are broken arbitrarily. Although k-NN is simply a truncated ball query, the real challenge lies in finding the proper search radius to enclose at least the n desired neighbors. To find the radius in practice depends on the choice of data structure used to access node locations.

A naïve approach for neighbor queries would be a brute-force search that checks distances from all nodes to every other node. Obviously the cost of such a method is high: $O(N^2)$ for all stencils. Multi-dimensional data structures, such as those discussed here, can limit the scope of searching and reduce the cost of stencil generation to $O(N \log N)$.

For the most part, investigations in RBF communities that delve into efficient neighbor queries are limited to ball queries. For example, the Partition of Unity method for approximation (e.g., [24, 25]), and particle methods like the Fast Multipole Method (e.g., [13, 26])

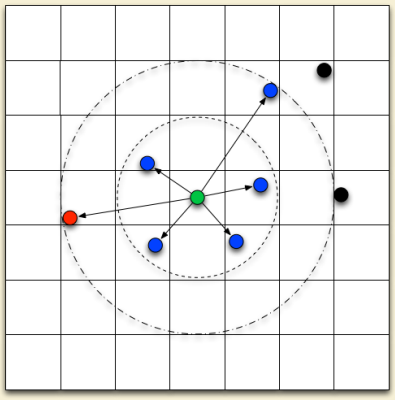


Figure 1.1: A stencil center in green finds neighboring stencil nodes in blue. Two ball queries are shown as dashed and dash-dot circles to demonstrate the added difficulty of finding the right query radius to obtain the k-nearest neighbors.

or Smoothed Particle Hydrodynamics (e.g., [15]). Examples of fast algorithms employed in these fields include the fixed-grid method [15, 25], k-D Trees [25], Range Trees [24, 25], and 2^d-Trees (i.e., Quad- and Octrees) [13, 26]. Surprisingly, while other communities continue the quest for fast neighbor queries, RBF collocation and RBF-FD communities have been slow on the uptake. For many years, the standard in the community has been to use k-D Trees (see e.g., [6, 8, 9]).

This chapter considers the use of an alternative neighbor query algorithm to generate RBF-FD stencils. It is based loosely on the fixed-grid method from [12, 14, 15]. [20] would classify the algorithm as a *fixed grid "bucket" method with one-dimensional spatial ordering*. The fixed-grid method loosens the requirements for finding the k-nearest neighbors (k-NN) stencils to accept k-"approximately nearest" neighbors (k-ANN). It also reorders nodes according to space-filling curves. In what follows, the fixed-grid method is compared to an efficient implementation of k-D Tree available for use in C++ and MATLAB ([22]). Benchmarks demonstrate that, with the proper choice of parameters for the fixed-grid, the method is up to 2x faster than k-D Tree, and it comes with a free bonus: up to 5x faster SpMV performance due to the impact of spatial reordering that occurs during stencil generation.

1.1 k-D Tree

A k-D Tree is a spatial data structure that generally decomposes a space/volume into a small number of cells. All k-D Trees are binary and iteratively subdivide volumes and sub-volumes at each level into two parts. The "k" in k-D Tree refers to the dimensionality of the data/volume partitioned—that is $k \equiv d$.

Given a set of points bounded by a d-dimensional volume, a k-D Tree applies a hierarchy of $(d-1)$-dimensional axis aligned *splitting planes* to cut the space. At each level of the hierarchy the splitting planes result in two new *half-planes* [21]. Consecutive splits intercept one another at a *splitting value*. k-D Trees do not require that half-planes equally subdivide

a volume; more often it is the data contained within the volume that is equally partitioned. The choice of dimension for the splitting plane, in conjunction with a variety of methods for choosing the splitting values allows for many flavors of k-D Trees (see e.g., [5, 20, 21] for comprehensive lists). *Point k-D Trees*, 2^d *Trees* (i.e., quad-/octrees), *BSP-Trees*, and *R-trees* are all members of the general k-D Tree class [21, 26].

This work considers *Point k-D Trees* [20], which partition a set of discrete points/nodes as outlined by the recursive procedure in Algorithm 1.1. Point k-D Trees assume that splitting planes intercept nodes rather than occur arbitrarily along the half-plane. The splitting value at each level of the tree is set to the *median coordinate* of the points in the half-plane, which ensures the tree is well balanced on initial construction. All nodes with coordinate (in the current dimension) less than or equal to the splitting value are contained by the left half-plane, and all nodes with coordinate greater than the splitting value are contained by the right. Half-planes containing only one element correspond to leaves of the tree. The median coordinate of a half-plane is found by sorting the n node coordinates contained by the partition and selecting the $\lceil \frac{n}{2} \rceil$-th element [5].

Algorithm 1.1 BuildKDTree(P, *depth*)

Input: A set of d-dimensional points P and the current *depth*.
Output: The root of the k-D Tree for P.

if $size(P) = 1$ **then**
    return a new leaf storing P
end if
$L_i := median(coord(P, depth))$
$v_l :=$ BuildKDTree($coord(P, depth) \leq L_i$, $(depth + 1)$ modulo d)
$v_r :=$ BuildKDTree($coord(P, depth) > L_i$, $(depth + 1)$ modulo d)
return A new node $v := \begin{pmatrix} \text{value} := L_i \\ \text{left} := v_l \\ \text{right} := v_r \end{pmatrix}$

The k-D Tree in Figure 1.2 is an example of a Point k-D Tree. Given a set of eight nodes in two dimensions, the tree is constructed by applying one-dimensional cuts along the x-dimension, then the y-dimension, then back to the x-dimension. This approach is referred to as *cyclic splitting*, as consecutive cuts are applied by iterating dimensions in a round-robin fashion [20]. The first cut, $L1$, shown in green, splits the nodes into two sets on either side of A. The corresponding tree in the center of Figure 1.2 shows $L1$ as the tree root with all nodes having x-coordinates less than or equal to A to the left, and all nodes having x-coordinates greater than A to the right. The second level of the tree, $L2$ and $L3$ (in blue), splits the half-planes on either side of A at nodes B and C. The axis parallel splits for each half-plane intercept $L1$ independently to partition half-planes along the y-dimension; once again, nodes with coordinates less than or equal (i.e., below) to the splitting value branch left in the tree, and y-coordinates greater than (i.e., above) the value branch right. The third level (red) returns to splitting half-planes in the x-dimension. Nodes D and H are not intersected by a splitting plane; their half-planes contain only one node so they immediately become leaves of the tree. This process to build a Point k-D Tree

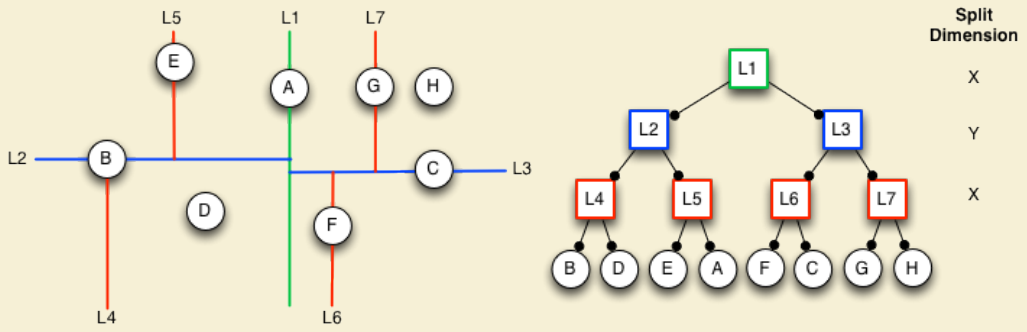


Figure 1.2: An example k-D Tree in 2-Dimensions. Nodes are partitioned with a cyclic dimension splitting rule (i.e., splits occur first in X, then Y, then X, etc.); all splits occur at the median node in each dimension.

has a complexity of $O(N \log N)$ with $O(N)$ storage required [5, 20].

Frequently, the terms *k-D Tree* and *Point k-D Tree* are used synonymously by the RBF community (see e.g., [6, 8, 9]); the same is convention adopted here.

Generating an RBF-FD stencil with a k-D Tree can be efficiently accomplished in $O(n \log N)$ time—where n is the stencil size—following an approach introduced in [10], and presented in Algorithm 1.2. The k-NN search starts a depth-first recursive search of the k-D Tree to find the nearest neighbor to a query point, X_q. Traversal of the the tree occurs by following branches left or right based on comparison of X_q coordinates to the splitting value stored at each node of the tree, with the objective to find the smallest half-plane containing X_q. The search traverses the height of the tree in $O(\log N)$ steps to find the leaf that stores the nearest neighbor to X_q. The neighbor point and its distance from X_q are inserted into a global priority queue, pq. Points in the priority queue are sorted in descending order according to distance.

After finding the nearest neighbor the algorithm returns to the previous split in the tree and traverses onto the opposing half-plane (i.e., down the far branch) to look for other leaves. So long as the size of pq is at less than capacity (n) the search automatically adds points to the priority queue. If pq reaches capacity the algorithm starts to pop off excess points with the understanding that the action removes those points farthest from X_q.

In order to prune branches from the search and reduce complexity, Algorithm 1.2 makes use of a routine called "BoundsOverlapBall", which checks if any boundaries of the current level half-plane intersect/overlap with a closed ball centered at X_q. The ball is given a radius equal to the maximum distance in pq. Then, if the ball and a boundary intersect, the search will continue onto the half-plane on the opposite side of that boundary. This step handles the possibility that nearer nodes occur within the overlapped region in the other half-plane. If the ball and boundary do not intersect, the opposing half-plane and its related subtree are pruned from the search. Additional details on the implementation of "BoundsOverlapBall" can be found in [10, 23].

The authors of [10] find Algorithm 1.2 capable of efficiently querying the n-nearest neighbors with a complexity proportional to $O(\log N)$ (dominated by the cost of tree traversal). The relationship between stencil size n, and grid size, N, is better expressed as $O(n \log N)$

Algorithm 1.2 KNNSearchKDTree(X_q, n, *root*, *depth*)

Input: A query node X_q, number of desired neighbors (n), the current *root* of the k-D Tree, and the current *depth* of traversal.
Output: A global priority queue, pq, containing the n-nearest neighbors to X_q sorted by distance from X_q in descending order.
Assume: A routine named "BoundsOverlapBall" exists to determine if the boundaries of the current half-plane are intersected by the ball centered at X_q with radius equal to the maximum distance in pq. As long as $pq.size < n$, "BoundsOverlapBall" defaults to true.

```
if root is leaf then
    Insert {root, dist(X_q, root)} into pq
    if pq.size > n then
        pq.pop                                  ▷ Keep only n-nearest neighbors
    end if
    return
end if

if coord(X_q, depth) <= root.value then
    KNNSearchKDTree(X_q, n, root.left, (depth + 1) % d)
else
    KNNSearchKDTree(X_q, n, root.right, (depth + 1) % d)
end if

if coord(X_q, depth) <= root.value then
    if BoundsOverlapBall(X_q) then
        KNNSearchKDTree(X_q, n, root.right, (depth + 1) % d)
    end if
else
    if BoundsOverlapBall(X_q) then
        KNNSearchKDTree(X_q, n, root.left, (depth + 1) % d)
    end if
end if
return
```

for one stencil.

RBF-FD only needs to generate stencils once, so the overall time for the step subsumes the cost of tree construction and N queries. The resulting total complexity of stencil generation for all stencils is then proportional to $O(N \log N)$.

1.2 A Fixed-Grid Algorithm

While a k-D Tree functions well for queries, the cost to build the tree structure is unnecessary overhead. Among the many data-structures that exist for nearest neighbor queries, alternatives like fixed-grid methods [20, 24, 25] (a.k.a. uniform grid [12, 15]) bypass

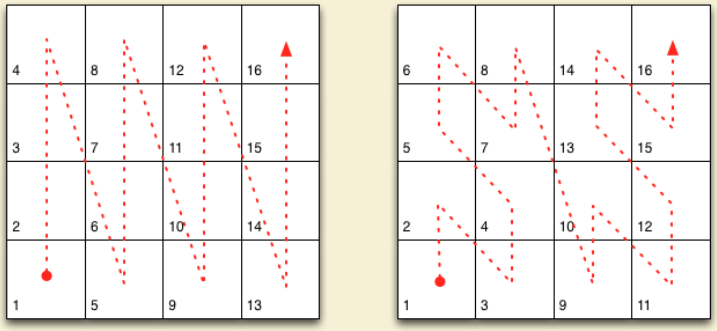


Figure 1.3: Two example space filling curves to linearize the same fixed-grid. Left: Raster-ordering (ijk); Right: Morton-/Z-ordering.

much of the cost in construction with an assumption that only lower spatial dimensions (e.g., 2-D or 3-D) are significant for choosing neighbors. This discards the need to build a tree and shifts focus onto querying neighbors.

Fixed-grid methods get their name from a coarse 2-D or 3-D regular grid that is overlaid on the domain. The d-dimensional grid divides the domain's axis aligned bounding box (AABB)—that is, the minimum bounding box containing the entire domain with edges parallel to axes—into $(h_n)^d$ cells. Subdivisions are uniform, so one can easily identify the cell containing any sample point, p, given the coordinates of the AABB and $(h_n)^d$. For example, let (c_x, c_y, c_z) be the desired cell in 3-D, and $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$ be the minimum and maximum coordinates of the AABB (resp.). Then the cell coordinates are found by:

$$\begin{aligned}(dx, dy, dz) &= \left(\frac{(x_{max} - x_{min})}{h_n}, \frac{(y_{max} - y_{min})}{h_n}, \frac{(z_{max} - z_{min})}{h_n}\right) \\ (c_x, c_y, c_z) &= \left(\left\lfloor \frac{(p_x - x_{min})}{dx} \right\rfloor, \left\lfloor \frac{(p_y - y_{min})}{dy} \right\rfloor, \left\lfloor \frac{(p_z - z_{min})}{dz} \right\rfloor\right). \end{aligned} \tag{1.1}$$

Cells neighboring (c_x, c_y, c_z) are trivial to find by adding positive and negative offsets to each coordinate.

Fixed grid methods also make use of *space filling curves*. Space filling curves pass through every point in d-dimensional space, and through each point only once. Equivalently, space filling curves map d-dimensional space down to 1-D, where every point is converted to a unique index or traversal order based on its spatial coordinates. These mapping properties make space filling curves ideal for use as hash functions. Traversing the d-dimensional points (i.e., playing "connect the dots") draws the space filling curve. Figure 1.3 presents two common orderings of a 2-D fixed-grid. Note that one-dimensional orderings are not unique. On the left is a *Raster*-ordering (a.k.a. Scanline- or ijk-ordering): $f(c_x, c_y, c_z) = ((c_x * h_n) + c_y) * h_n + c_z$. The right half of Figure 1.3 shows an ordering known as Morton- or Z-ordering. Z-ordering construction is discussed later in this chapter. On both sides of Figure 1.3, the lower left corner of each cell indicates the mapped index. Traversing the cells in order produces the curves superimposed in red.

At a high level, fixed-grid methods have the following construction steps [15]:

1. Subdivide the domain with the overlay grid.

2. For each node, identify the containing cell coordinates.

3. For each node, use the cell coordinates as input to a spatial hash function (i.e., a space-filling curve).

4. Sort the nodes according to their spatial hash.

Particular details of how nodes are sorted, the choice of hashing function, the number of nodes allowed per cell, etc. determine the specific class of fixed-grid method and corresponding complexity. A comprehensive list of options and classifications can be found in [20].

1.2.1 Fixed-grid Construction

The algorithm in this work is inspired by fixed-grid approaches for GPU particle simulations ([12, 14, 15]). Particle methods require a ball query at each time-step. With time-steps often dominated by the cost of querying neighbors, the community understandably devotes significant effort to seek out the most efficient solutions possible [11]. The fixed-grid method is competitive for at least two reasons: a) by bypassing the need to build a tree, half the cost in querying neighbors is avoided; and b) nodes sorted according to a spatial hash reside closer in memory to nearby neighbors than in the case of unsorted nodes. The spatial locality results in a higher likelihood that data will be cached when required. Note that reordering by cell hash sorts nodes across cells but not within them—that is, nodes contained by the same cell are contiguous in memory, but remain arbitrarily ordered with respect one another. Fortunately, with contiguous groups of nodes, nearest neighbor queries can directly access all nodes per cell.

The authors of [12, 14, 15] assume a raster-ordering on cells, and that the uniform grid is sufficiently refined to ensure cells contain at most eight nodes. Particle interactions are limited to ball queries on the containing cell plus one valence of neighboring cells (i.e., 8 surrounding cells in 2-D and 26 cells in 3-D). Since the number of cells to check is fixed, the neighboring nodes can be obtained by direct access in constant time. A similar approach is taken by [11], but the authors opt for a Z-ordering of cells. As a point of difference in implementations, the authors of [11, 12, 15] leverage a fast radix sort algorithm to order nodes based on hash index, while [14] utilizes a slower bitonic sort algorithm. The fixed-grid in [24, 25] forgoes logic to refine the grid and enforce a maximum limit on the number of nodes per cell. The author also avoids sorting nodes based on cell hashes. Instead, a list is maintained that stores the indices of all contained nodes per cell.

The implementation presented here is a hybrid of the related algorithms. For example, cells are sorted based on raster-ordering, but without the restriction on max number of nodes per cell. Rather than a radix- or bitonic sort to reorder nodes, the list of node indices for each cell ([24, 25]) is constructed as part of a single-pass bucket sort. Finally, in stark contrast to [12, 14, 15, 24, 25], querying neighbors is not restricted to a fixed radius, or number of cell valences. To satisfy the k-NN query, this implementation iteratively increases the query radius to include a new valence of cells at each iteration. This multi-pass ball-

query was demonstrated in Figure 1.1. The iteration terminates when the desired count of neighboring nodes is satisfied or exceeded.

Algorithm 1.3 BuildFixedGrid(P, h_n)

Input: A set points P, and the fixed-grid resolution, h_n.
Output: The reordered points in P, and corresponding cell buckets Q.

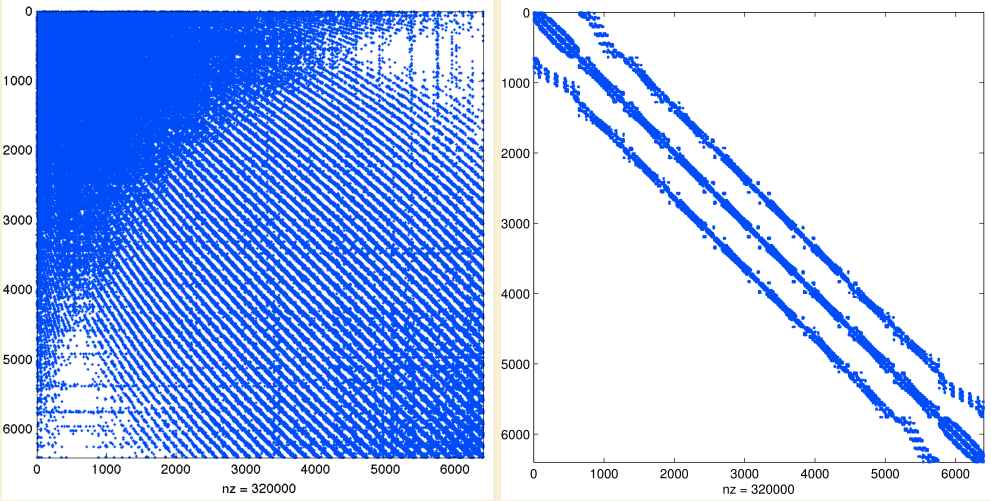
Create Q: an $(h_n)^d$ array of empty buckets.
for point p_i in P **do**
    $c :=$ CellCoords(p_i)
    $ind :=$ SpatialHash(c)
    Append index i onto $Q[ind]$
end for
for $j = 0, 1, ..., (h_n)^d$ **do**
    if $Q[j]$ is not empty **then**
        Append the set $P[Q[j]]$ onto $\hat{P}$
        Overwrite the set $Q[j]$ with new indices of $\hat{P}$
    end if
end for
$P := \hat{P}$
return

Algorithm 1.3 presents the fixed-grid build process. The routine starts with the allocation an array of empty buckets, Q. Next Q is populated based on the spatially hashed cell coordinates. The second for-loop in Algorithm 1.3 iterates through Q, looking for non-empty buckets. When one is found, nodes referenced by that bucket are transcribed/appended onto the "sorted" list of nodes $\hat{P}$. This way the nodes in each cell are contiguous, but maintain the original ordering with respect to one another. Additionally, node indices in $\hat{P}$ replace the old indices within Q.

The entire build process complexity is proportional to $O(N)$, and requires $O((h_n)^d + N)$ storage. Samet [20] would classify this approach as a *fixed-grid bucket method with one-dimensional ordering*. The term *bucket* refers to the allowance for each cell to contain an arbitrary number of nodes. *One-dimensional ordering* is indicative of attempts later in the chapter to employ alternative space-filling curves in place of raster-ordering.

A special note: the final step of Algorithm 1.3 overwrites the original list of nodes with the sorted equivalent. The spatially sorted list is included in the cost of stencil generation, but available for reuse elsewhere. Since the first step in RBF-FD applications is to generate stencils, overwriting the input node set can guarantee that the node values throughout the entire life-cycle of an RBF-FD application will benefit from the same spatial locality as stencil generation. This benefit is (almost) free.

Consider Figure 1.4, which shows two differentiation matrices generated based on the same $N = 6400$ MD-node set (unit sphere), with each row representing an RBF-FD stencil of $n = 50$ non-zeros. The left matrix in Figure 1.4 is generated with stencils queried by a k-D Tree. The k-D Tree maintains the original ordering on P. The matrix on the right of Figure 1.4 is a permuted equivalent of the left matrix with fixed-grid cells sorted based on a raster-ordering for $h_n = 10$.



(a) k-D Tree (b) Fixed-Grid

Figure 1.4: Example effects of node reordering for MD node set $N = 6400$ with $n = 50$. The differentiation matrices are permuted equivalents and roughly 0.78% full. a) Stencils generated based on k-D Tree maintain the original node ordering. b) The reordered node set generated using an $h_n = 10$ fixed-grid condenses non-zeros for improved memory access patterns (i.e., cache reuse).

Looking at Figure 1.4 it should be obvious that reordering the nodes can improve memory access patterns for SpMV. If each row is applied as a sparse dot-product with a dense vector, the more condensed non-zeros are in the row, the more likely values from the dense vector will be resident in cache when needed. Likewise, non-zeros that appear on consecutive rows can benefit from cache reuse. Later in this chapter, the impact of spatial orderings are compared to determine how RBF-FD can benefit the most.

1.2.2 Fixed-Grid Neighbor Query

Querying the k-nearest neighbors for a single query node, X_q, is the subject of Algorithm 1.4. The process begins by finding X_q's containing cell, c. The hash value of c is used to identify (in Q) the list of nodes contained within that cell, all of which are appended to a vector of neighbor candidates, pq.

It is possible for the number nodes in c to exceed n; however, the algorithm conservatively assumes that it is necessary to search at least one valence of cells around it. This ensures that nodes near the cell boundaries will find nearby neighbors outside of c, and the stencils will balanced. For certain fixed-grid resolutions, a single valence may not satisfy the stencil size requirements, so the algorithm iterates outward onto new valences. As cells are checked, their nodes are appended onto pq.

The final stage of Algorithm 1.4 calculates the distance from all candidate nodes to X_q, and uses that metric to sort pq in ascending order. The first n nodes in pq are returned as the stencil.

Complexity of Algorithm 1.4 can vary based on the choice of h_n. For a sufficiently

Algorithm 1.4 QueryFixedGrid(X_q, n, P, Q)

Input: A query point, X_q; the desired number of neighbors, n; a set of d-dimensional points P; and the matching cell bucket list, Q.
Output: The n-nearest neighbors list pq.

$valence := 1$
$c :=$ CellCoords(X_q)
$ind :=$ SpatialHash($cells$)
Append $P[Q[ind]]$ onto pq
while pq.size $< n$ OR $valence < 2$ **do**
    $cells :=$ NeighboringCellCoords($c, valence$)
    $inds :=$ SpatialHash($cells$)
    for each q in $Q[inds]$ **do**
        if q is not empty **then**
            Append node list $P[q]$ onto pq
        end if
    end for
    increment $valence$
end while
$dists :=$ ComputeDistances(pq)
Sort pq by $dists$
return the first n nodes in pq

refined fixed-grid the k-ANN is dominated by the cost of the *while-loop* and behaves as $O(\log h_n)$ per stencil. In the worst case, when h_n is small, the cost of sorting pq dominates, and is proportional to $O(N \log N)$ (using a C++ STL Sort) in the worst case. Results below demonstrate that proper choice of h_n can maintain logarithmic complexity similar to the k-D Tree query.

The fixed grid query algorithm is considered an *approximate nearest neighbor* (ANN) search. Consider again the nodes in Figure 1.1. A k-NN stencil of size $n = 8$ should contain the green center, all blue nodes, plus the red node and one black node. The true k-NN would select the black node in the right-most column of the grid (i.e. the node closer to the dashed ball query). Under the fixed-grid method, however, the alternative black node is selected even though it is more distant. This is true because the more distant black node occupies the second valence of cells around the stencil center, whereas the true near neighbor is in the third valence. Algorithm 1.4 is able to truncate the search in the second valence by satisfying the requirement on n.

Similarly, if cells are rectangular in shape, the ball-query under fixed-grid functions as an ellipsoid. In this case, stencils are biased with more nodes in one direction. To combat this, and ensure spherical stencils, this work assumes the AABB bounding the domain is a cube (i.e., $dx = dy = dz$).

The difference between a true k-NN and k-ANN is insignificant from the perspective of RBF-FD. The method compensates automatically for differences in node locations when weights are calculated. On top of this, the only differences in stencils generated by k-NN versus k-ANN occur at nodes on the outermost reaches of the stencil (i.e., nodes with the

least impact on the stencil center).

1.2.3 Performance Comparison

The implementation of k-D tree compared in this work, the *kd-Tree Matlab* library, was originally posted to the Matlab FileExchange in 2008 [22] and now maintained as an independent Google Code project [23]. The implementation is written in C++, but includes a MEX compiled interface, allowing for a consistent and efficient k-D Tree API in both languages. The original release of *kd*-Tree Matlab (pre-2012) was in use throughout the RBF community at the onset of this work. The dual language API was appealing for rapid-prototyping with MATLAB, and then porting applications to C++.

The pre-2012 implementation followed the $O(N \log N)$ expected complexity for neighbor queries, but the cost to build the tree was another story. Build times for small and medium sized grids (i.e., less than $N = 50000$ nodes) were small enough to be inconspicuous. However, large grid sizes were found to be prohibitively high, with the implementation behaving as $O(N^2)$. This issue ultimately led to work on a MATLAB prototype of the fixed-grid method ([3]) to test the concept of neighbor queries with low build costs.

The fixed-grid implementation, originally a pure MATLAB script, outperformed the "efficient" MEX-compiled k-D Tree for problem sizes $N > 20000$, and lead to the development of a faster C++ implementation tested here. However, with the 2012 release of *kd*-Tree Matlab on Google Code ([23]), the author has significantly improved the performance of the build process to achieve the $O(N \log N)$ behavior expected for a Point k-D Tree with cyclic splitting.

All benchmarks in this section were performed on the Itasca cluster at the Minnesota Supercomputing Institute. Itasca is an HPLinux cluster with 1,134HP ProLiant blade servers, each with two-socket, quad-core 2.8 GHz Intel Xeon processors sharing at least 24GB of RAM [1].

Figure 1.5 compares the performance of the k-D Tree and the fixed-grid method for increasing regular grid resolutions. Both the current k-D Tree implementation (2012) and the original implementation (pre-2012) are shown on the left. For comparison, the fixed-grid method is shown with two overlay resolutions: $h_n = 50$ and $h_n = 100$. Associated speedups are shown in The log-log plot reveals that the

performance gap between the C++ fixed-grid implementation and the k-D Tree. Prior to the 2012 improvements the C++ implementation was over 130x faster in stencil generation (build and query). However, with the more reasonable

Figure 1.5 compares the total time to generate N stencils of size $n = 50$ with three methods: [**?**], our hash-based neighbor query, and the improved KDTree from [**?**]. Until the new release of KDTree, our algorithm was a major improvement to the performance of stencil generation. The hash-based approach achieved greater than

The difficulty in choosing h_n should not deter the use of fixed-grids for RBF-FD. Since stencil generation is a one time cost, so long as the fixed-grid method is on the same order of complexity as k-D Tree, the true cost is insignificant. The long-run payoff on SpMV performance is where the fixed-grid method wins the argument over choice in method.

[2]

what is best overlay resolution? based on time to generate. choose resolution as n/2,

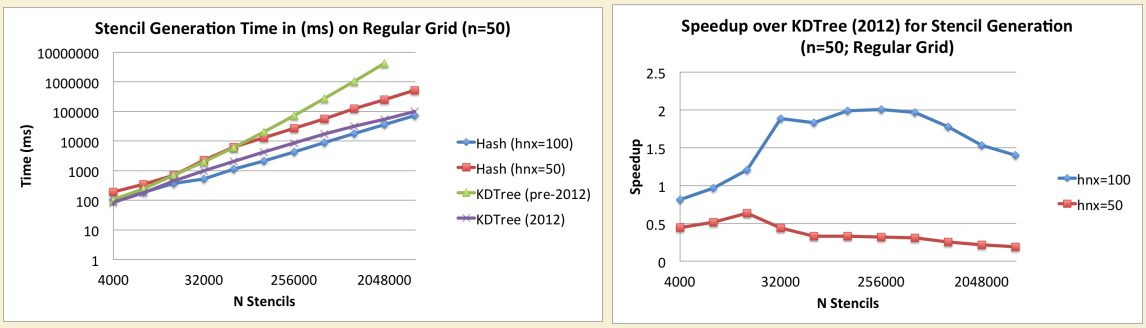


Figure 1.5: Querying the $n = 50$ nearest neighbors on a regular grid up to $N = 160^3$ demonstrates the gains achieved by the fixed-grid neighbor query method.

n/9? etc?

The early implementation of k-D tree had an $O(n^2)$ growth in complexity. This algorithm was developed to alleviate that cost. It took a few hours to implement but has had some surprising impacts. Note that the complexity of k-D tree was reduced to $O(N \log^2 N)$ in 2012. On a regular grid (generated with raster/IJK ordering), the cost of k-D tree grows at the same rate as the hashing method.

At $N = 32000$ the cost of hashing drops below k-D Tree due to the decreasing number of empty hash cells. Likewise, at $N = 1000000$ and beyond, the gap between hashing and k-D Tree begins to close as cells contain more than one

Q: why does the curve drop for $hnx = 100$? Q: the complexity of the algorithm? Q: the sphere I understand: its localizing the search to small patches on the sphere, and

For every N there is an optimal hnx. This is depicted for $N = 500000$ CVT and $hnx = 100$.

1.2.4 Alternative Orderings

As mentioned, the choice of space filling curve can impact the sparsity pattern of differentiation matrices. Those patterns in turn correlate to differenct memory access patterns

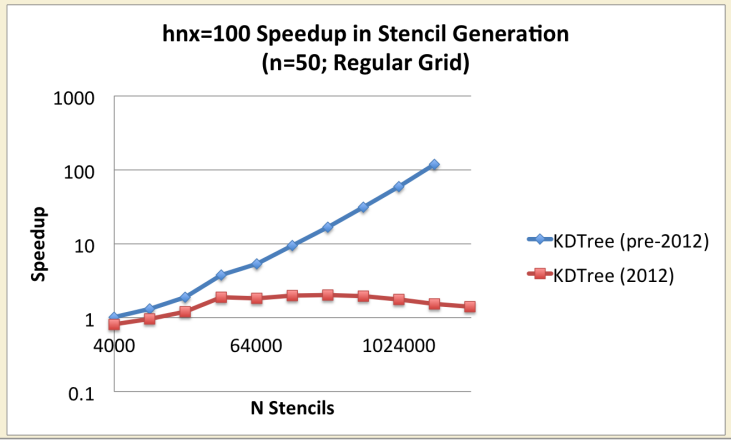
Examples include Z- or *Morton* ordering, U- or *Greycode*-ordering, X-ordering, etc. Hilbert curves have been demonstrated to be the most efficient [**?**].

The authors of [11] utilize Z-ordering in their

The ideal differentiation matrix, corresponding to discretization of a line, would have all non-zeros on the first $\frac{n-1}{2}$ to each side of the diagonal. For 2- or 3-D domains the ideal case is not possible to achieve, but the nodes can be reordered with the goal to condense At the end of BuildFixedGrid the original set of nodes, P is overwritten by $\hat{P}$. While nothing requires that $\hat{P}$ replace the usage of P

[17] found that reodering nodes via RCM and space filling curves offer similar benefits in terms of reduced TLB misses and better cache coherency.

Cuthill-McKee. The Cuthill McKee algorithms can be equated to a breadth-first search. The algorithm queues nodes in order of degree at each level of the search and traverses the lowest degree priority. The Reverse variant of Cuthill-McKee inverts the node order so that the lowest degree and top level node are at the end of the matrix rather than



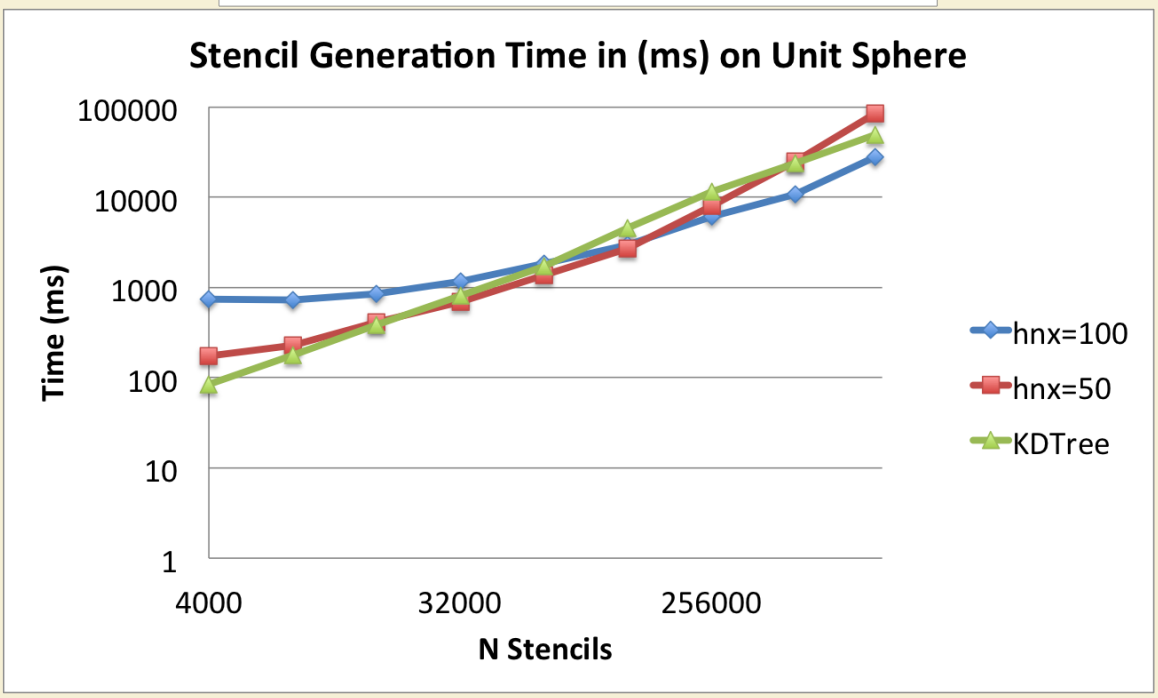


Figure 1.6: Querying the $n = 50$ nearest neighbors on a regular grid up to $N = 160^3$ demonstrates the significant gains achieved by our spatially binned neighbor query. While KDTree queries grow as $O(NlogN)$

the beginning. Aside from ordering, the Reverse and Standard Cuthill McKee algorithms are identical processes. RCM is the more popular of the variants though, due to storage savings and reduced fill-in for some decompositions [16].

1.2.5 Impact of Orderings

Rather than complete a full analysis of the impact from space filling curves, we only consider the ideal cases from CVT and RG (because they are used in the thesis). The integer dilation is costly and does not appear to significantly improve the performance of the best cases.

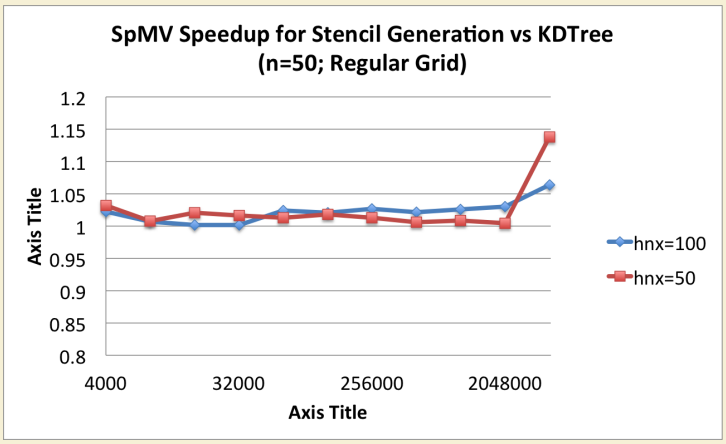


Figure 1.7: Querying the $n = 50$ nearest neighbors on a regular grid up to $N = 160^3$ demonstrates the significant gains achieved by our spatially binned neighbor query. While KDTree queries grow as $O(NlogN)$

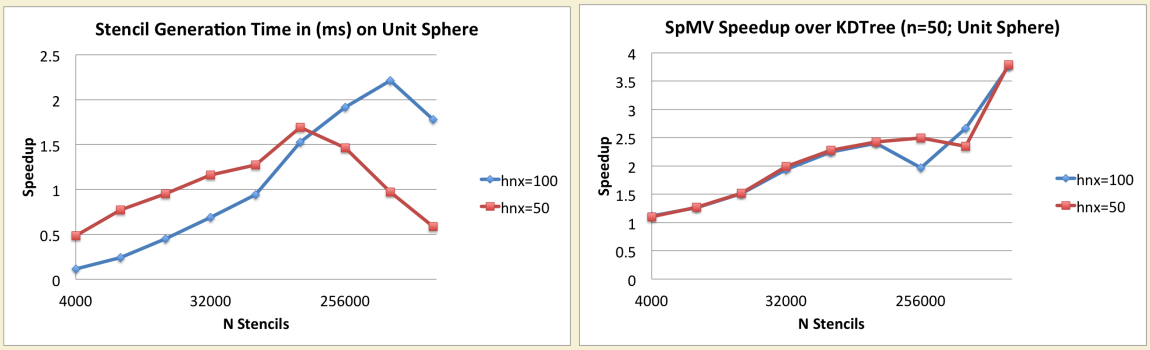


Figure 1.8: Generating stencils for increasing subsets of the $N = 1e6$ CVT nodes mesh.

RCM is effective and used with KDTree.

In Figure 1.16 typical stencils in RBF-FD easily span multiple cells. Shown is $n = 31$ which spans 7 cells. The use of Z-ordering (e.g., [11]) is often associated with improved memory access patterns. However,

The RCM algorithm is a breadth first search over stencils to order them based on the minimum degree (number of connections per stencil). The zig-zag ordering shown in Figure 1.16 is an ideal case where all nodes are connected to two neighbors (i.e., $n = 3$). In practice the curve trends diagonally across the domain due to the BFS property, but the ordering often appears to connect nodes randomly

Obviously, the ideal case for bandwidth is when all rows contain the $\frac{n}{2}$ nodes corresponding to solution value to either side of u_j. In 1-D this corresponds to every node containing the $\frac{n}{2}$ nodes to the left and right of x_j. In 2-D this is only possible if the nodes in the domain are properly indexed such that stencils contain the proper set of neighbors—a stringent requirement that will

impact from ordering on matrix sparsity. Bandwidth impact. Bandwidth impact on

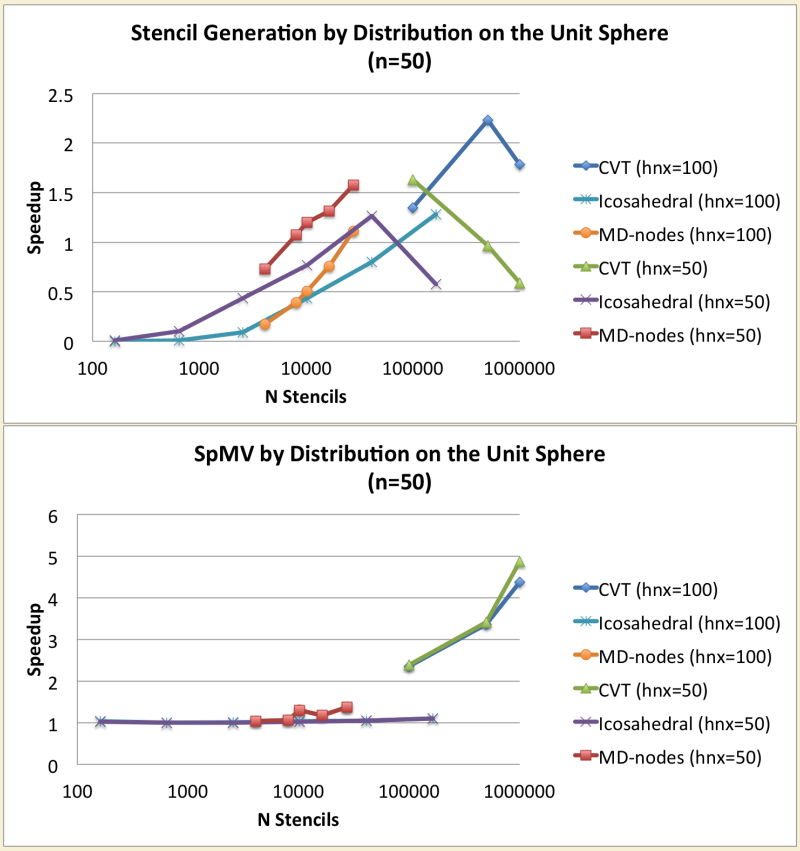


Figure 1.9: Based on the proper choice of overlay resolution, the hash stencil query can accelerate stencil generation, but the sophistication of the algorithm is low enough that negative impact is more likely. On the other hand, the impact on SpMV performance is always positive with the routine accelerated up to 4.9x faster.

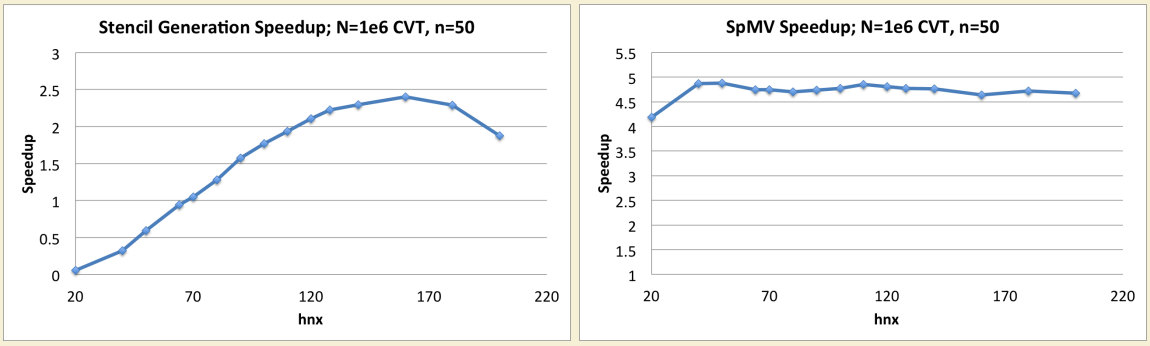
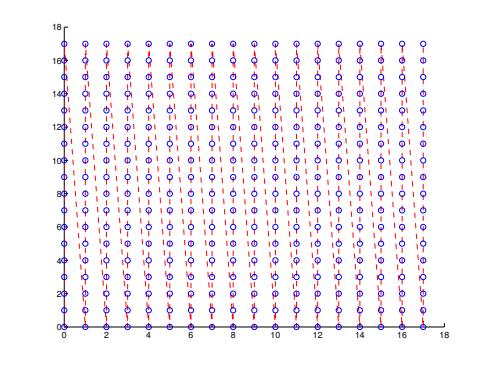
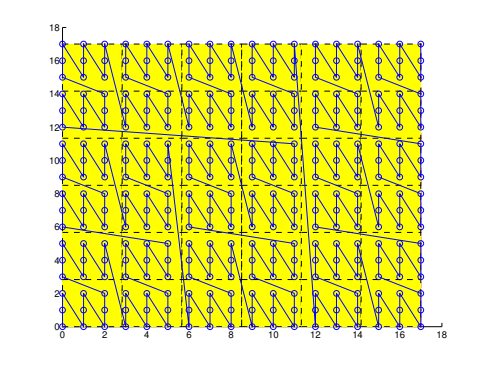


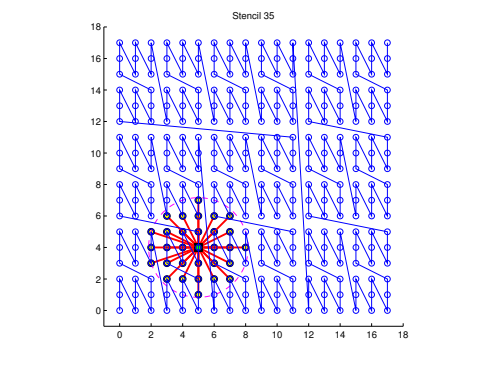
Figure 1.10: As the coarse grid resolution increases the hashing algorithm achieves both 2x faster than KDTree in stencil generation, with greater than 4x gain in SpMV performance (for free).



(a) Original Node Distribution



(b) Fixed-grid cells sorted by Z-order



(c) Example Stencil $n = 31$

Figure 1.11: In order: a) node ordering test cases; b) original ordering of regular grid (raster); c) coarse grid overlay for hash functions ($hnx = 6$); d) example stencil ($n = 31$) spanning multiple Z's; e) spy of DM after orderings.

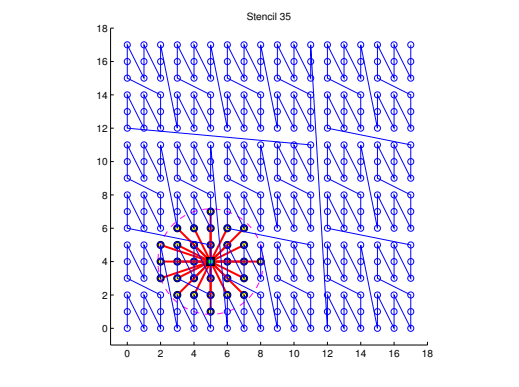


Figure 1.12: A simple example illustrates the use of a fixed-grid with $h_n = 6$ for $N = 18^3$ regular nodes. d) example stencil ($n = 31$) spanning multiple Z's; e) spy of DM after orderings.

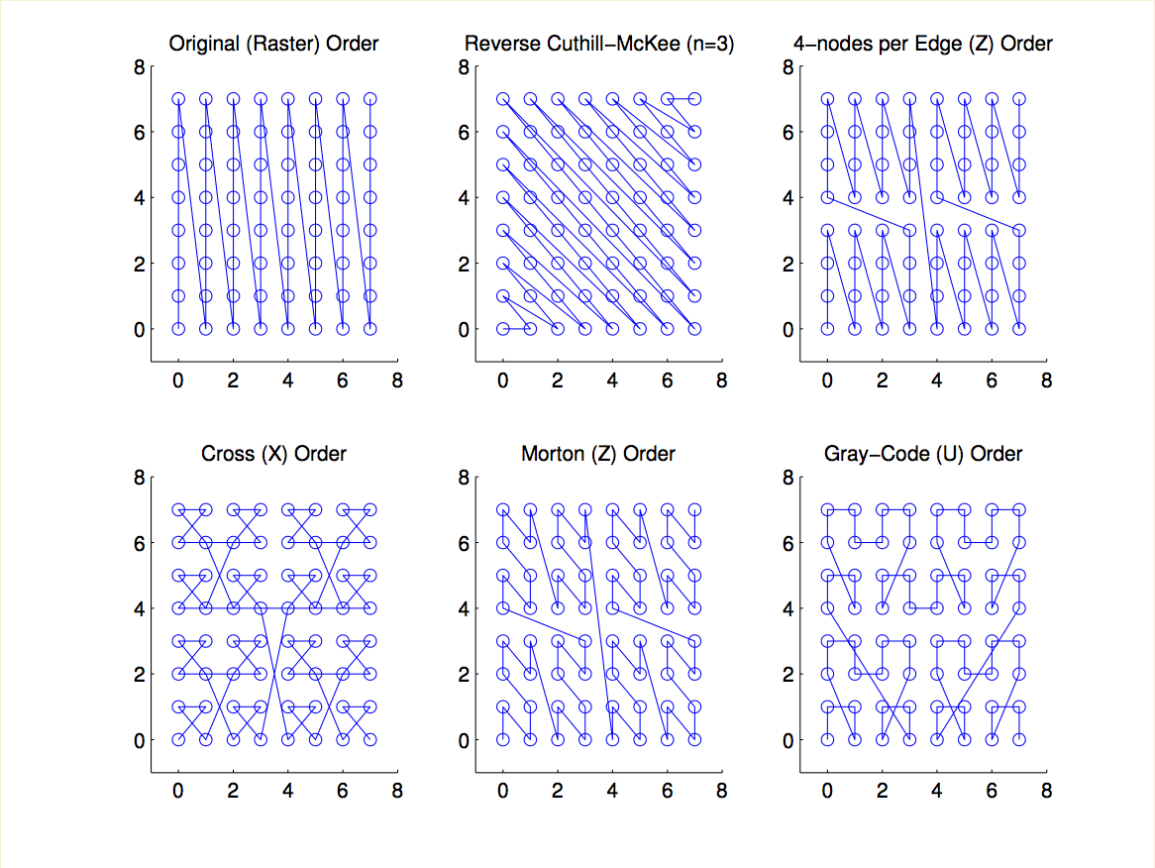


Figure 1.13: Example space filling curves used to reorder nodes. The Raster, Z, X, U and 4-nodes per Edge are space filling curves applied to reorder cells of the fixed-grid stencil queries. Reverse Cuthill-McKee (RCM) reorders nodes based on the output stencils. RCM shown here is a special case: each node has 3 neighbors.

condition considered in future chapter.

One frequently hears that ordering via space filling curves like Morton Ordering and/or gray codes can benefit memory access patterns.

(Related? http://publish.uwo.ca/~shaque4/presentationSONAD.pdf http://www.cs.duke.edu/~alvy/papers/sc98/ http://www.cs.indiana.edu/~dswise/Arcee/Papers/medea06.pdf http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.118.7720&rep=rep1&type=pdf http://stackoverflow.com/questions/4260002/benefits-of-nearest-neighbor-search

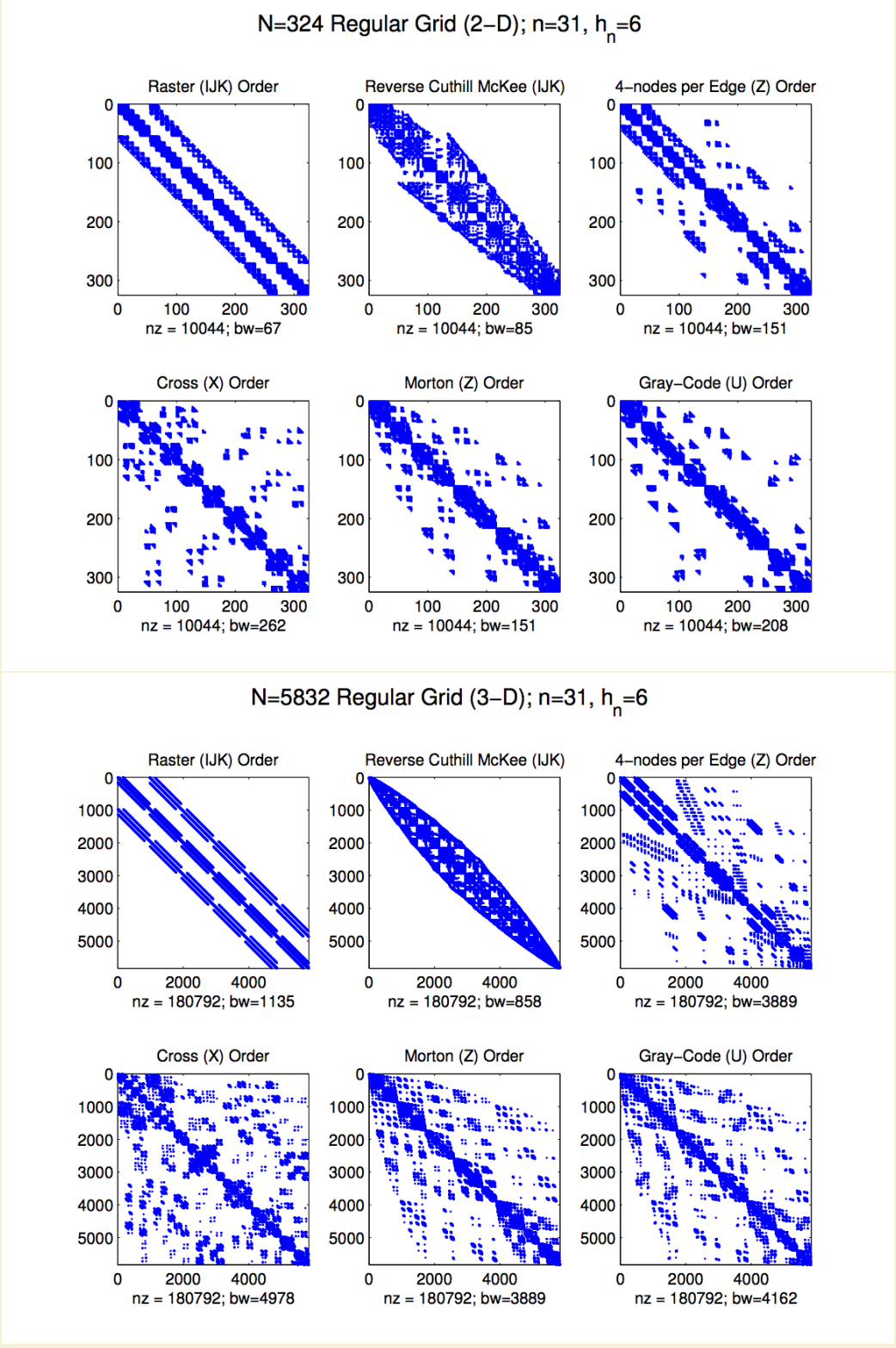


Figure 1.14: In order: a) node ordering test cases; b) original ordering of regular grid (raster); c) coarse grid overlay for hash functions ($hnx = 6$); d) example stencil ($n = 31$) spanning multiple Z's; e) spy of DM after orderings. Author's Note: TODO: Raster on top left. Add RCM ordering to top right. 3) Spy impact on MD node set $N = 4096$, stencil size $n = 31$, $h_n = 6$

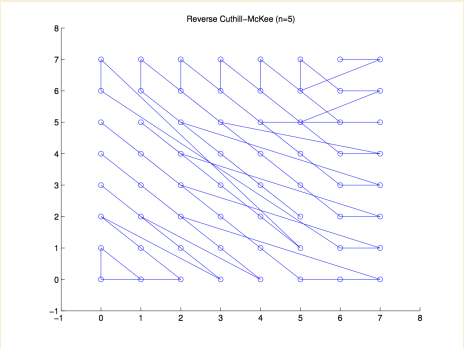


Figure 1.15: An RCM reordering of nodes based on stencils $n = 5$ nearest neighbors per node.

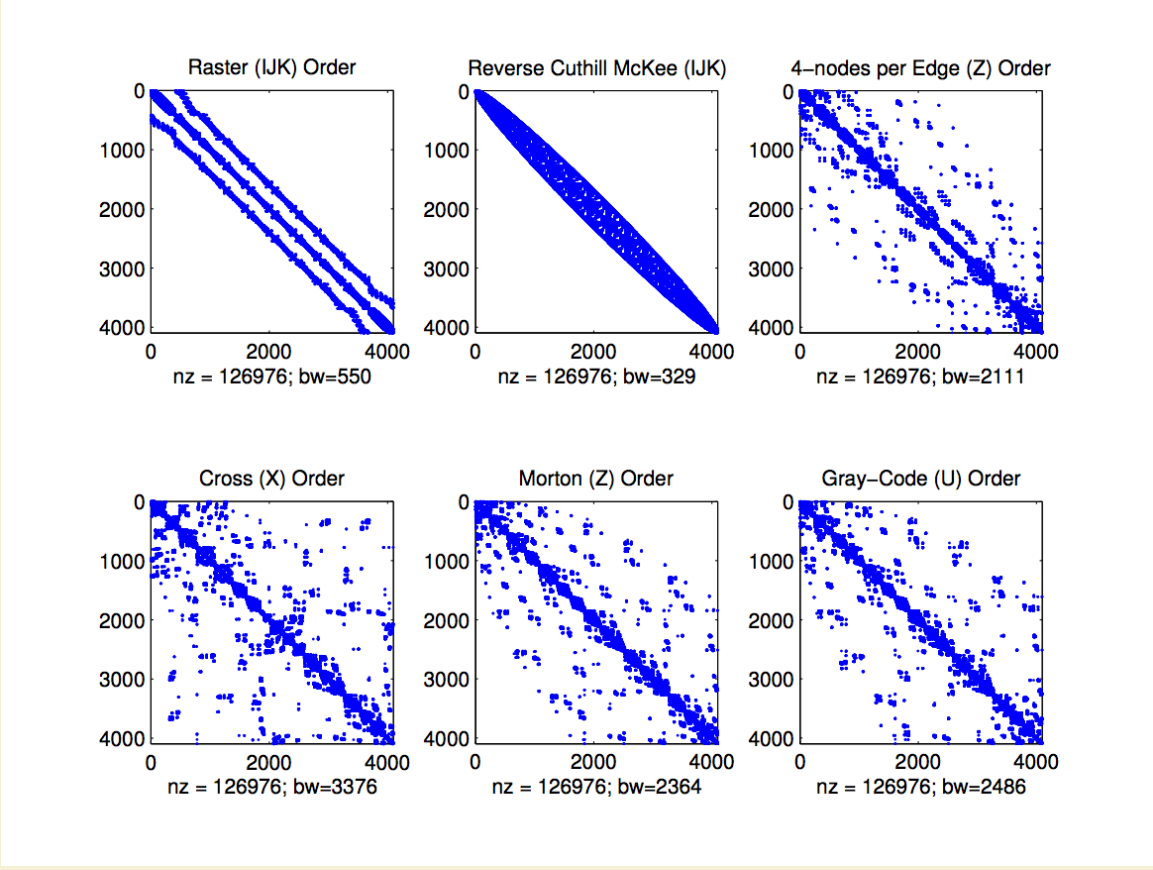


Figure 1.16: The impact of reordering on sparsity patterns for $N = 4096$ MD nodes on the sphere. Stencil size $n = 31$, fixed-grid resolution per dimension $h_n = 10$.

1.3 Conclusion and Future Work

Although RBF-FD only requires neighbor queries once, the results that follow reveal a long lasting positive impact on memory with a fixed-grid method, which is sufficient to justify its use. Investigations into moving node coordinates and/or local refinements for RBF methods (e.g., [7]) would find the fixed-grid method significantly more beneficial. As of this writing no known applications of RBF-FD consider moving nodes

Due to the limited significance of stencil generation under RBF-FD, the overhead in implementing and debugging the fixed-grid method on the GPU is difficult to justify. The implementation tested here was developed as a pure CPU prototype with minimal attention to optimization. The added complexity in reproducing the efficient fixed-grid method on the GPU could be the subject of future work for moving nodes.

[4] could benefit the algorithm by sorting nodes completely based on floating point Z-ordering.

1.3.1 Future Work

While more efficient implementations are possible, the savings demonstrated by Figure **??** the savings are significant with the right choice of h_n. Generating stencils for RBF-FD is a preprocessing cost, so we do not dedicate an excessive amount of attention to this algorithm. However, a few ideas that would improve: hilbert ordering, choose AABB resolution based on N not user parameters, faster sorting, GPU implementation

Open item from to this section include:

- An efficient GPU k-NN implementation for the investigation of RBF-FD in a moving nodes or adaptive mesh refinement situation. Examples include [4, 11, 18]
- Space filling curves are limited to a MATLAB prototype at the moment. These can be implemented in C++ for integers, with a potential investigation into floating point dilation and orderings [4].
-

[19] mentions the impact of ordering on conditioning.

1.4 Conclusions on Stencil Generation

For quasi-regular distributions and small to medium sized grids the k-D Tree performs well enough in comparison to the fixed-grid method. However, the difference in

There is an ideal h_n.

TODO:

- Performance of k-D Tree vs fixed-grid on regular and sphere grids.
  - conclude that the fixed-grid method is worth its weight but needs the proper choice of h_n.
  - add data to show what $h_n(N)$ should be.

  - state that benchmarks were performed on Itasca and what the hardware specs are.

- Node orderings based on integer dilation
  - Dilation and interleave process (include description of groups by dimension)
  - Give formulas for interleaving
  - RCM algorithm high level (?)
  - bandwidth implications on memory access (lower bandwidth is usually better)
  - table showing the impact of orderings on bandwidths

BIBLIOGRAPHY

[1] Itasca (hp proliant bl280c g6 linux cluster). https://www.msi.umn.edu/hpc/itasca, June 2013. 11

[2] Kdtreesearcher class. MATLAB R2013a Documentation (http://www.mathworks.com/help/stats/kdtreesearcherclass.html), Aug 2013. 1, 11

[3] E. F. Bollig. A fixed-grid prototype for rbf stencil generation with various space filling curves. https://github.com/bollig/rbf_hash_query, 2012. 11

[4] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Transactions on Visualization and Computer Graphics*, 16:599–608, 2010. 20

[5] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008. 3, 4

[6] Gregory E. Fasshauer. *Meshfree Approximation Methods with MATLAB*, volume 6 of *Interdisciplinary Mathematical Sciences*. World Scientific Publishing Co. Pte. Ltd., 5 Toh Tuck Link, Singapore 596224, 2007. 2, 4

[7] Natasha Flyer and Erik Lehto. Rotational transport on a sphere: Local node refinement with radial basis functions. *Journal of Computational Physics*, 229(6):1954–1969, March 2010. 20

[8] Natasha Flyer, Erik Lehto, Sebastien Blaise, Grady B. Wright, and Amik St-Cyr. Rbf-generated finite differences for nonlinear transport on a sphere: shallow water simulations. *Submitted to Elsevier*, pages 1–29, 2011. 2, 4

[9] Bengt Fornberg and Erik Lehto. Stabilization of RBF-generated finite difference methods for convective PDEs. *Journal of Computational Physics*, 230(6):2270–2285, March 2011. 2, 4

[10] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, September 1977. 4

[11] Prashant Goswami, Philipp Schlegel, Barbara Solenthaler, and Renato Pajarola. Interactive sph simulation and rendering on the gpu. In *Proceedings of the 2010 ACM*

SIGGRAPH/Eurographics Symposium on Computer Animation, SCA '10, pages 55–64, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association. 7, 12, 14, 20

[12] S. Green. Cuda particles. NVidia Whitepaper, 2010. 2, 5, 7

[13] N. A. Gumerov, R. Duraiswami, and E. A. Borovikov. Data structures, optimal choice of parameters, and complexity results for generalized multilevel fast multipole methods in d dimensions. Technical Report UMIACS-TR-2003-28, University of Maryland (College Park, Md.), Apr 2003. 1, 2

[14] I. Johnson. Real-time particle systems in the blender game engine. Master's thesis, Florida State University, November 2011. 2, 7

[15] Øystein E. Krog. GPU-based Real-Time Snow Avalanche Simulations. Master's thesis, Norwegian University of Science and Technology, June 2010. 2, 5, 6, 7

[16] Wai-Hung Liu and Andrew H. Sherman. Comparative analysis of the cuthill-mckee and the reverse cuthill-mckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, 13(2):pp. 198–213, Apr 1976. 13

[17] John Mellor-crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. In *International Journal of Parallel Programming*, pages 425–433, 2001. 12

[18] Jia Pan and Dinesh Manocha. Fast GPU-based Locality Sensitive Hashing for K-Nearest Neighbor Computation. *Proceedings of the 19th ACM SIGSPATIAL GIS '11*, 2011. 20

[19] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial Mathematics, second edition, 2003. 20

[20] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. 2, 3, 4, 5, 7, 8

[21] Steven Skiena. *The Algorithm Design Manual (2. ed.)*. Springer, 2008. 2, 3

[22] Andrea Tagliasacchi. kd-tree for matlab. `http://www.mathworks.com/matlabcentral/fileexchange/21512-kd-tree-for-matlab`, Sep 2010. 1, 2, 11

[23] Andrea Tagliasacchi. kd-tree matlab. `https://code.google.com/p/kdtree-matlab`, Jun 2012. 4, 11

[24] Holger Wendland. Fast evaluation of radial basis functions: Methods based on partition of unity. In *Approximation Theory X: Wavelets, Splines, and Applications*, pages 473–483. Vanderbilt University Press, 2002. 1, 2, 5, 7

[25] Holger Wendland. *Scattered Data Approximation*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2005. 1, 2, 5, 7

[26] Lexing Ying. A kernel independent fast multipole algorithm for radial basis functions. *Journal of Computational Physics*, 213(2):451 – 457, 2006. 1, 2, 3