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

ABSTRACT

Many numerical methods based on Radial Basis Functions (RBFs) are gaining popularity in the geosciences due to their competitive accuracy, functionality on unstructured meshes, and natural extension into higher dimensions. One method in particular, the Radial Basis Function-generated Finite Differences (RBF-FD), has drawn significant attention due to its comparatively low computational complexity versus other RBF methods, high-order accuracy (6th to 10th order is common), and parallel nature.

Similar to classical Finite Differences (FD), RBF-FD computes weighted differences of stencil node values to approximate derivatives at stencil centers. The method differs from classical FD in that the test functions used to calculate the differentiation weights are n-dimensional RBFs rather than one-dimensional polynomials. This allows for generalization to n-dimensional space on completely scattered node layouts.

Although RBF-FD was first proposed nearly a decade ago, it is only now gaining a critical mass to compete against well known competitors in modeling like FD, Finite Volume and Finite Element. To truly contend, RBF-FD must transition from single threaded MATLAB environments to large-scale parallel architectures.

Many HPC systems around the world have made the transition to Graphics Processing Unit (GPU) accelerators as a solution for added parallelism and higher throughput. Some systems offer significantly more GPUs than CPUs. As the problem size, N, grows larger, it behooves us to work on parallel architectures, be it CPUs or GPUs. In addition to demonstrating the ability to scale to hundreds or thousands of compute nodes, this work introduces parallelization strategies that span RBF-FD across multi-GPU clusters. The stability and accuracy of the parallel implementations are tested in explicit and implicit modes to verify correctness.

This work establishes RBF-FD as a contender in the arena of distributed HPC numerical methods.

CHAPTER 1

AN ALTERNATIVE STENCIL GENERATION ALGORITHM FOR RBF-FD

Like all RBF methods, RBF-FD is designed to handle irregular node distributions, and 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 like the sphere [6, 7, 8, 9] and quadrilateral/prism [**?**] 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 each node, including the query node. This is known more formally as a *k-nearest neighbor (k-NN)* problem [24] (a.k.a. ℓ-nearest neighbor search [27]). Here "nearest" is defined with the Euclidean distance metric, although it is possible to generalize to other metrics (see e.g., [1]).

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 [27]) due to the closed ball created by the radius of support in 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 k 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 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 approx-

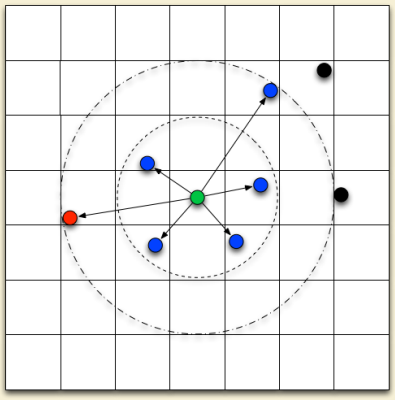


Figure 1.1: A stencil center in green finds neighboring stencil nodes in blue. Two ball queries are shown as dashed and dash-dot circles to demonstrate the added difficulty of finding the right query radius to obtain the k-nearest neighbors.

imation (e.g., [26, 27]), and particle methods like the Fast Multipole Method (e.g., [14, 28]) or Smoothed Particle Hydrodynamics (e.g., [16]). Examples of fast algorithms employed in these fields include the fixed-grid method [16, 27], k-D Trees [27], Range Trees [26, 27], and 2^d-Trees (i.e., Quad- and Octrees) [14, 28]. 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., [5, 7, 10]).

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 [13, 15, 16]. [22] 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 ([24]). 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 due to the impact of spatial reordering that occurs by default 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* [23]. 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., [4, 22, 23] 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 [23, 28].

This work considers *Point k-D Trees* [22], 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 [4].

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 [22]. 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 the cut) 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

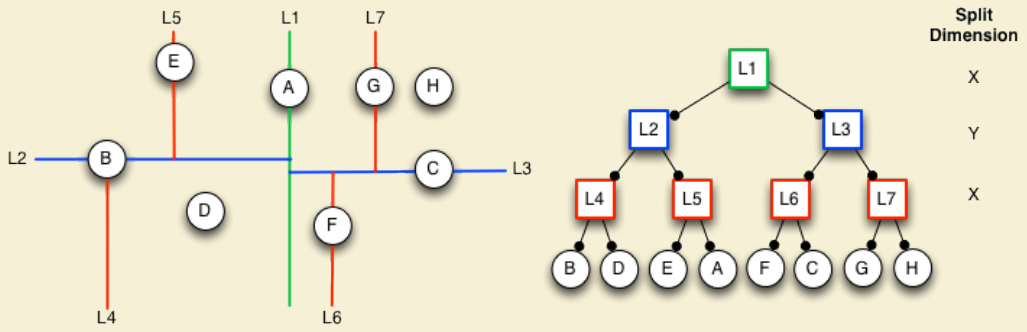


Figure 1.2: An example k-D Tree in 2-Dimensions. Nodes are partitioned with a cyclic dimension splitting rule (i.e., splits occur first in X, then Y, then X, etc.); all splits occur at the median node in each dimension.

node so they immediately become leaves of the tree. This process to build a Point k-D Tree has a complexity of $O(N \log N)$ with $O(N)$ storage required [4, 22].

Frequently, the terms *k-D Tree* and *Point k-D Tree* are used synonymously by the RBF community (see e.g., [5, 7, 10]); 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 [11], 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 node in the tree and 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 [11, 25].

The authors of [11] 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).

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
```

The relationship between stencil size n, and grid size, N, is better expressed as $O(n \log N)$ 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 overall complexity is 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 [22, 26, 27] (a.k.a. uniform grid [13, 16]) bypass 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 containing 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 [16]:

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 [22].

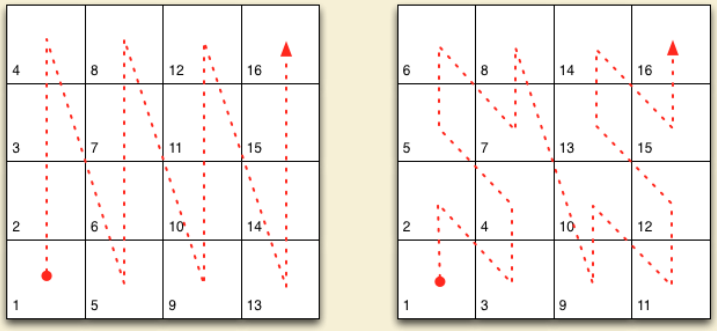


Figure 1.3: Two example space filling curves to linearize the same fixed-grid. Left: Raster-ordering (ijk); Right: Morton-/Z-ordering.

1.2.1 Fixed-grid Construction

The algorithm in this work is inspired by fixed-grid approaches for GPU particle simulations ([13, 15, 16]). 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. 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 [13, 15, 16] 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. As a point of difference in implementations, the authors of [13, 16] leverage a fast radix sort algorithm based on hash index to order nodes, while [15] utilizes a slower bitonic sort algorithm. The fixed-grid in [26, 27] 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 for each cell that has the indices of all contained nodes, which implies random memory access when operating on the nodes of a 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 ([26, 27]) is constructed as part of a single-pass bucket sort. Finally, in stark contrast to [13, 15, 16, 26, 27], querying neighbors is not restricted to a fixed 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.

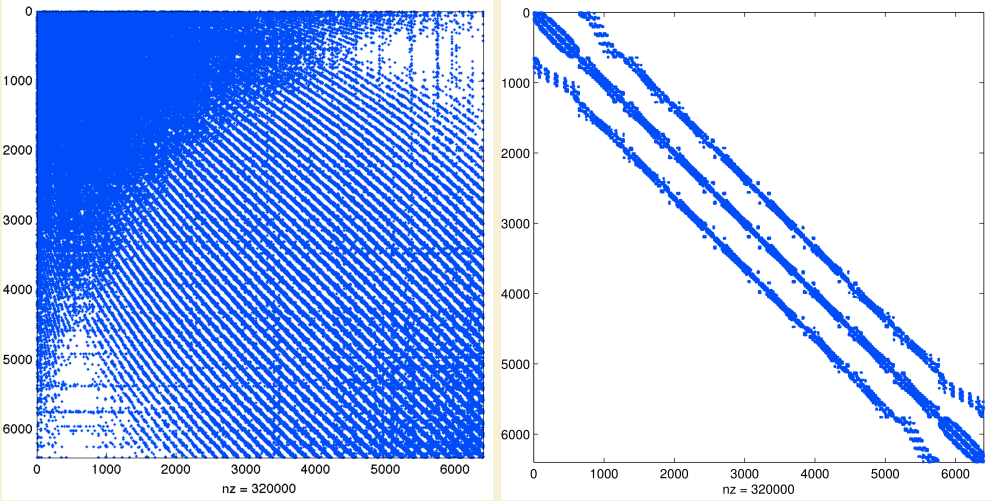
Create Q: an $(h_n)^d$ array of empty buckets.
for point p_i in P **do**
    $c :=$ CellCoords(p_i)
    $ind :=$ SpatialHash(c)
    Append index i onto $Q[ind]$
end for
for $j = 0, 1, ..., (h_n)^d$ **do**
    if $Q[j]$ is not empty **then**
        Append the set $P[Q[j]]$ onto $\hat{P}$
        Overwrite 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 [22] 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 RBF-FD stencils 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 stencils connecting the exact same nodes. In this case, the fixed-grid cells have a raster-based ordering with $h_n = 10$.



(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 cache effects.

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. 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 k-NN

Querying the k-nearest neighbors from the fixed-grid is the subject of Algorithm 1.4. First, the query node is hashed to identify its cell. Second

1.2.3 Orderings

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

1.2.4 k-NN with Fixed-Grid

Start at the center cell and add the 8 cells immediately surrounding it. Add to this the 16 cells in the second halo for a total of 25 cells.

So while a k-NN stencil of size $n = 8$ in Figure 1.1 would contain the black node in the right-most column of the grid. The k-ANN generated a stencil of size $n = 8$ would opt for

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)
Append $Q[c]$ 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
    $valence := valence + 1$
end while
$dists :=$ ComputeDistances(pq)
Sort pq by $dists$
return the first n nodes in pq

the black node to the top right because it is in the second halo and accepted as "closer" because the stencil can be satisfied without querying the extra 24 cells on the outermost halo.

Therefore, it is not essential that stencils contain only nearest neighbors. Instead, one can acquire the *approximate nearest neighbors*. Figure **??** demonstrates a case where a does not contain all nearest neighbors. As illustrated in the Figure, the ANN stencil and true nearest neighbor stencil differ by one node. THis is not dire

The query is approximate because nodes such as the white node in Figure 1.1 could be included in the stencil before

1.2.5 Performance

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., [6]) 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.

[16] assumes that at each time-step a new virtual fixed grid is generated

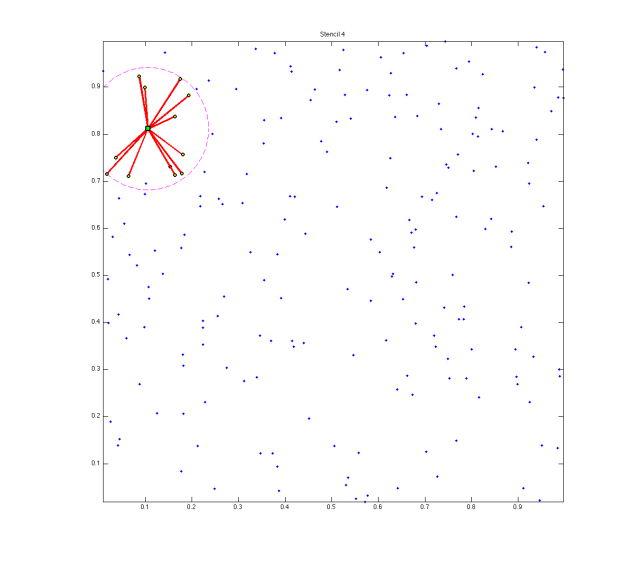


Figure 1.5: A stencil generated with k-ANN satisfies the required stencil size, but is not guaranteed to choose the true nearest neighbors.

Implementations of fixed-grid often assume that no more than two nodes reside in any one cell.

This implementation is CPU only. Porting the one time stencil generation to the GPU was seen as

use of fixed-grid methods originated due to low overhead before performing queries. Although all three related works focus on offloading simulations to the GPU, the method is also popular for CPU implementations

In [26] and later [27], Wendland presents the fixed-grid method as a low-cost alternative for RBF methods based on Partition of Unity.

[20] adopt the a similar approach that incrementally grows the

Following [22], the method used in this work is classified as a fixed-grid *bucket* method with one dimensional ordering. The emphasis on bucket arises from the application of a coarse grid that allows

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

The algorithm chosen for this work is provided in Algorithm **??**. The algorithm starts by creating a list of lists representing the $(h_n)^d$ cells. As nodes are matched with cell indices, the node index is appended to the

These tests assume $dx = dy = dz$ to ensure spherical stencils.

The implementation runs entirely on the CPU ([16] runs on the GPU).

The k-D tree implementation compared in this work is the *kd-Tree Matlab* library from Tagliasacchi [24, 25]. Originally posted to the Matlab FileExchange and now maintained as a Google Code project. Until the addition of $KNNsearcher$ in Matlab

garnered attention due to its MEX compiled interface that allowed use as both a C++ and MATLAB library. The Matlab central k-D Tree is MEX compiled and efficient. We

integrated the standalone C++ code into our library.

Tagliasacchi [25] uses the standard split by median in each dimension, with alternating chosen by incrementing the dimension by one, modulo the number of dimensions (e.g., in 2D the order is $X, Y, X, Y, ...$). Wendland [27] suggests a number of more advanced median cuts such as

[24]. [1]

1.2.6 Integer Dilation and Node Reordering

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

[22] labels this type of algorithm as a *fixed grid bucket algorithm* with the twist that it also has *one dimensional ordering*

1.3 Performance Comparison

At the start of this dissertation, the most widely used k-D Tree implementation in the RBF community was [24]. As a MEX-compiled

- Performance improvement
- SpMV impact
- impact from space filling curves

Many algorithms exist to query the k-nearest neighbors (equivalently all nodes in the minimum/smallest enclosing circle). Some algorithms overlay a grid similar to Locality Sensitive Hashing and query such as... [**?**].

For the purpose of generating RBF-FD stencils the kD-Tree is built and all queries are made once. There is no reuse of the tree so the build and query times are combined into one.

impact from ordering on matrix sparsity. Bandwidth impact. Bandwidth impact on condition considered in future chapter.

what is best overlay resolution? based on time to generate. choose resolution as n/2, n/9? etc?

perform neighbor queries on raster order because its easiest to jump cells. then each cell can be reordered according to z,x,u, etc. for better memory locality. Matlab script to do this (can be ported to C)

http://www.vincentgarcia.org/data/Garcia_2010_ICIP.pdf

GPU version of Locality Sensitive Hashing could reduce complexity further [19]

This can be done efficiently using neighbor query algorithms or spatial partitioning data-structures such as Locality Sensitive Hashing (LSH) and kD-Tree. Different query algorithms often have a profound impact on the DM structure and memory access patterns. We choose a Raster (ijk) ordering LSH algorithm [**?**] leading to the matrix structure in Figures **??** and **??**. While querying neighbors for each stencil is an embarrassingly parallel operation, the node sets used here are stationary and require stencil generation only once.

Efficiency and parallelism for this task has little impact on the overall run-time of tests, which is dominated by the time-stepping. We preprocess node sets and generate stencils serially, then load stencils and nodes from disk at run-time. In contrast to the RBF-FD view of a static grid, Lagrangian/particle based PDE algorithms promote efficient parallel variants of LSH in order to accelerate querying neighbors at each time-step [12, 19].

At the onset of our work on RBF-FD, the most commonly used KDTree implmentation used by the RBF community was [**?**]. Recently, improvements were made to the KDTree algorithm to reduce the cost of building the KDTree to $O(Nlog^2N)$.

Figure **??** 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

Our work in [2] leveraged an alternative to k-D tree, based loosely on space-filling curve orderings common in Lagrangian schemes like Smoothed Particle Hydrodynamics (e.g., [**?**], [**?**]).

1.3.1 Future Work

[**?**] introduce a minimal k-D Tree for the GPU which

The complexity of the method is still higher than the more efficient implementations used by Lagrangian methods, but as demonstrated in Figure **??** the savings are significant. 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

RBF-FD operates on general node distributions. Historically, stencils are uniform in size (n) and generated by selecting the $(n-1)$ true nearest neighbors to a node x_c. This is a k-NN query.

Alternative queries are possible: ball query and approximate nearest neighbor. The approximate is of particular interest because nodes closest to the stencil will always be selected, whereas the nodes further away have minimal influence so swapping out cant hurt. The justification in altering the selection is for reduced complexity in neighbor queries.

1.3.2 Hashing

[3] provide a fast parallel

[**?**] is working on parallel generation. [**?**] has OpenCL neighbor queries

We started with a CPU implementation to test appropriateness.

Approximate nearest neighbors will be nearly balanced. We observe that RBF-FD functions as well on stencils of true nearest neighbors as it does on approximate nearest neighbors.

To demonstrate the savings in choice of stencil generation method, we provide Figure **??**.

The impact of our neighbor query also extends influence on the structure of the RBF-FD DMs. has is to To quantify the sparsity of a Differentiation Matrix we consider the ratio of non-zeros ($N * n$) to total elements in the matrix (N^2). For example, a problem of size $N = 10,000$ with stencil size $n = 31$ has a ratio of 0.0031 and is 99.69% empty.

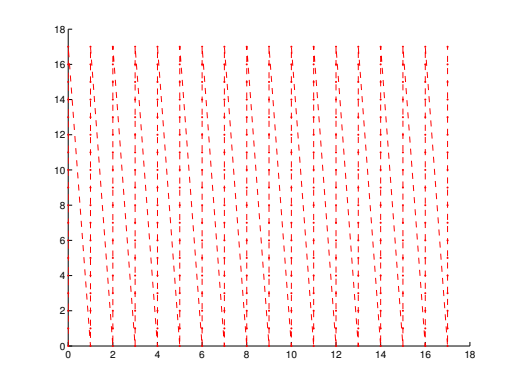


Figure 1.6: In order: a) node ordering test cases; b) original ordering of regular grid (raster); c) coarse grid overlay for hash functions ($hnx = 6$); d) example stencil ($n = 31$) spanning multiple Z's; e) spy of DM after orderings. Author's Note: REGENERATE FIGURES WITH RANDOM/HALTON NODES

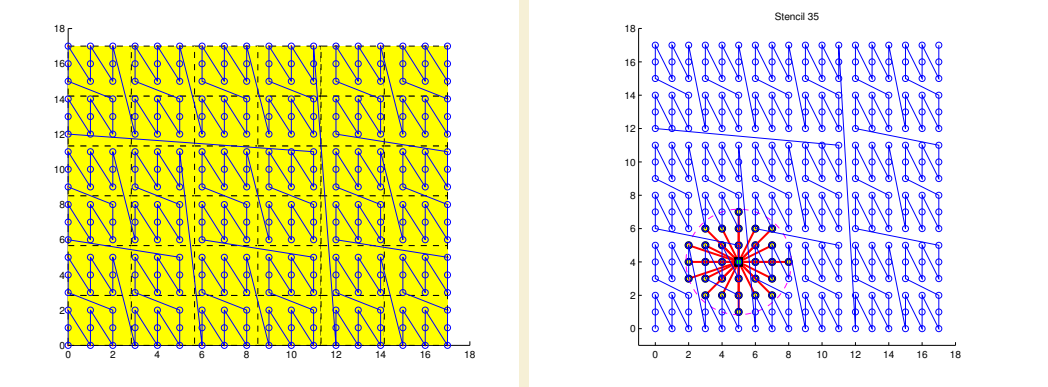


Figure 1.7: In order: a) node ordering test cases; b) original ordering of regular grid (raster); c) coarse grid overlay for hash functions ($hnx = 6$); d) example stencil ($n = 31$) spanning multiple Z's; e) spy of DM after orderings.

Querying neighbors requires searching at least the immediate cell one layer of neighbors. by including one extra layer we ensure that small stencils near the border of the immediate cell can pick up neighbors in adjacent cells.

A prototype implementation of this method allowed for a variety of space filling curves to reorder the cells. The curves are created through integer dilation [**?**]

The KDTree implementation used in this work is from

Original data showed our algorithm as wildly successful against a version

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 the beginning. Aside from ordering, the Reverse and Standard Cuthill McKee algorithms are identical processes.

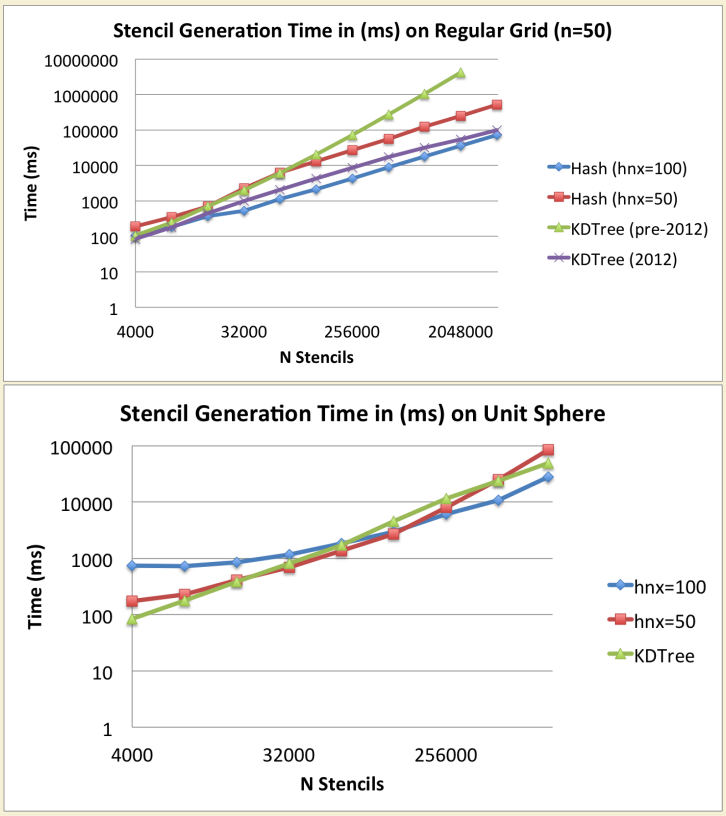


Figure 1.8: Querying the $n = 50$ nearest neighbors on a regular grid up to $N = 160^3$ demonstrates the significant gains achieved by our spatially binned neighbor query. While KDTree queries grow as $O(NlogN)$

RCM is the more popular of the variants though, due to storage savings and reduced fill-in for some decompositions [17]. .

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

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

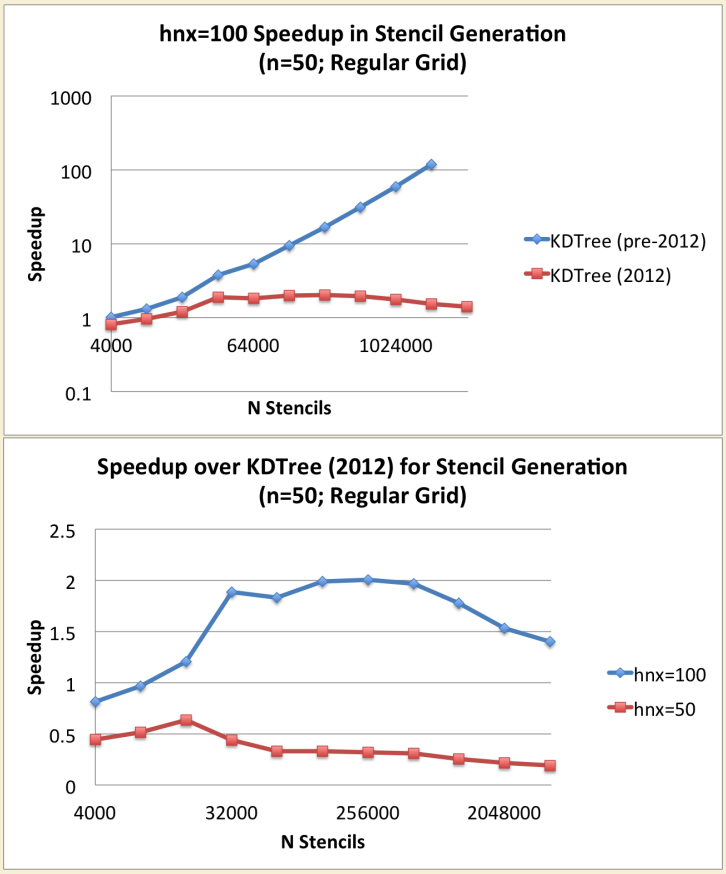


Figure 1.9: Querying the $n = 50$ nearest neighbors on a regular grid up to $N = 160^3$ demonstrates the significant gains achieved by our spatially binned neighbor query. While KDTree queries grow as $O(NlogN)$

$hnx = 100$.

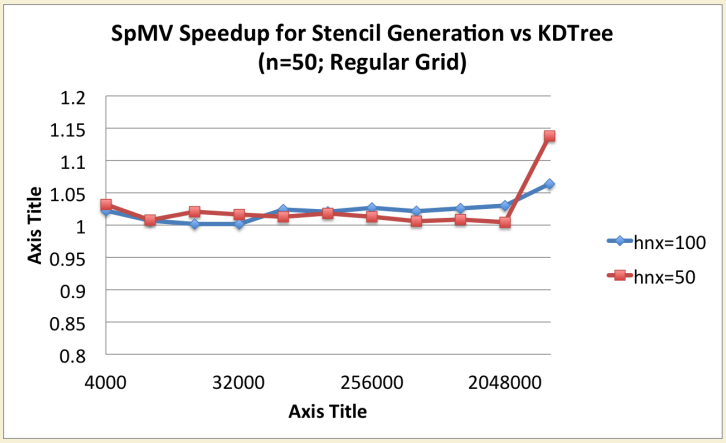


Figure 1.10: Querying the $n = 50$ nearest neighbors on a regular grid up to $N = 160^3$ demonstrates the significant gains achieved by our spatially binned neighbor query. While KDTree queries grow as $O(NlogN)$

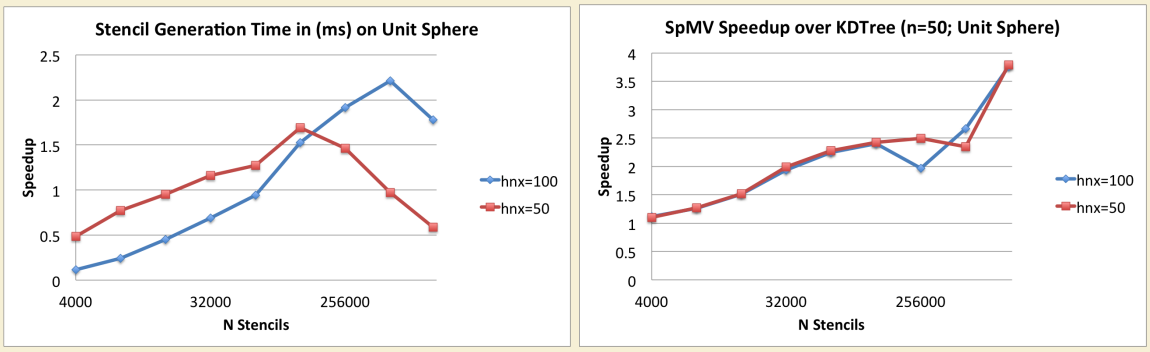


Figure 1.11: Generating stencils for increasing subsets of the $N = 1e6$ CVT nodes mesh.

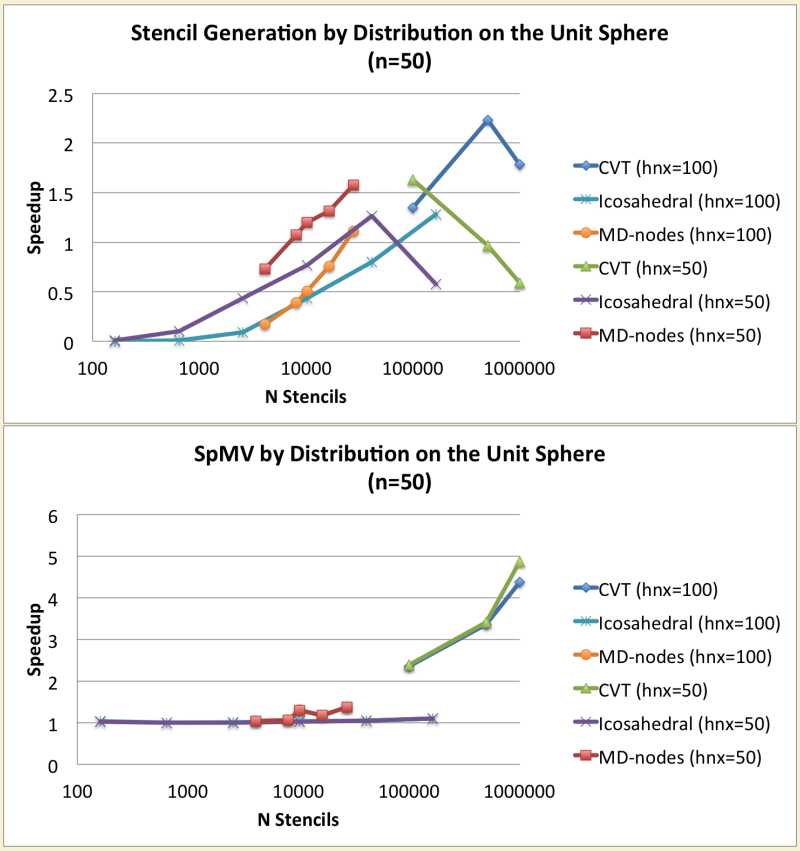


Figure 1.12: Based on the proper choice of overlay resolution, the hash stencil query can accelerate stencil generation, but the sophistication of the algorithm is low enough that negative impact is more likely. On the other hand, the impact on SpMV performance is always positive with the routine accelerated up to 4.9x faster.

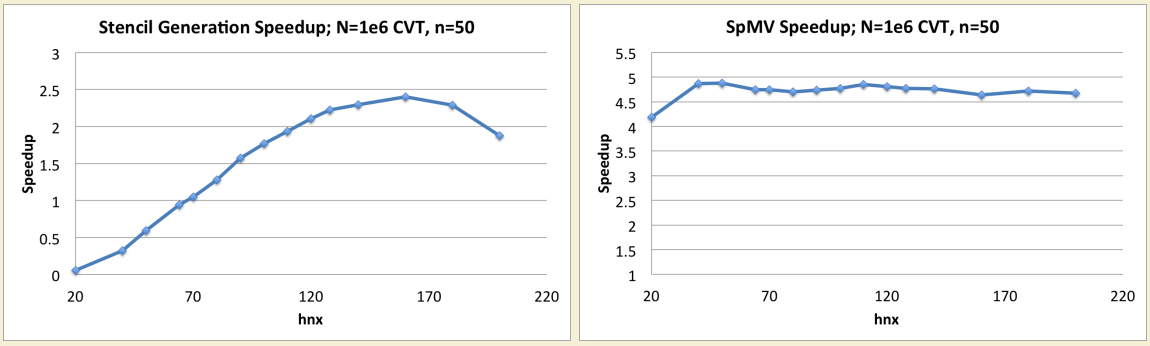


Figure 1.13: As the coarse grid resolution increases the hashing algorithm achieves both 2x faster than KDTree in stencil generation, with greater than 4x gain in SpMV performance (for free).

1.4 On Space Filling Curves and Other Orderings

1.4.1 Integer Dilation

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

[21] mentions the impact of ordering on conditioning.

Algorithms like Reverse Cuthill McKee and Approximate Minimum Degree ordering allow general restructuring of matrices.

Author's Note: NEed to compare conditioning of LSH and other algorithms in Matlab

Q: what is an ideal ordering? Q: what is the best conditioning from ordering? Q: what is the relative cost of ordering?

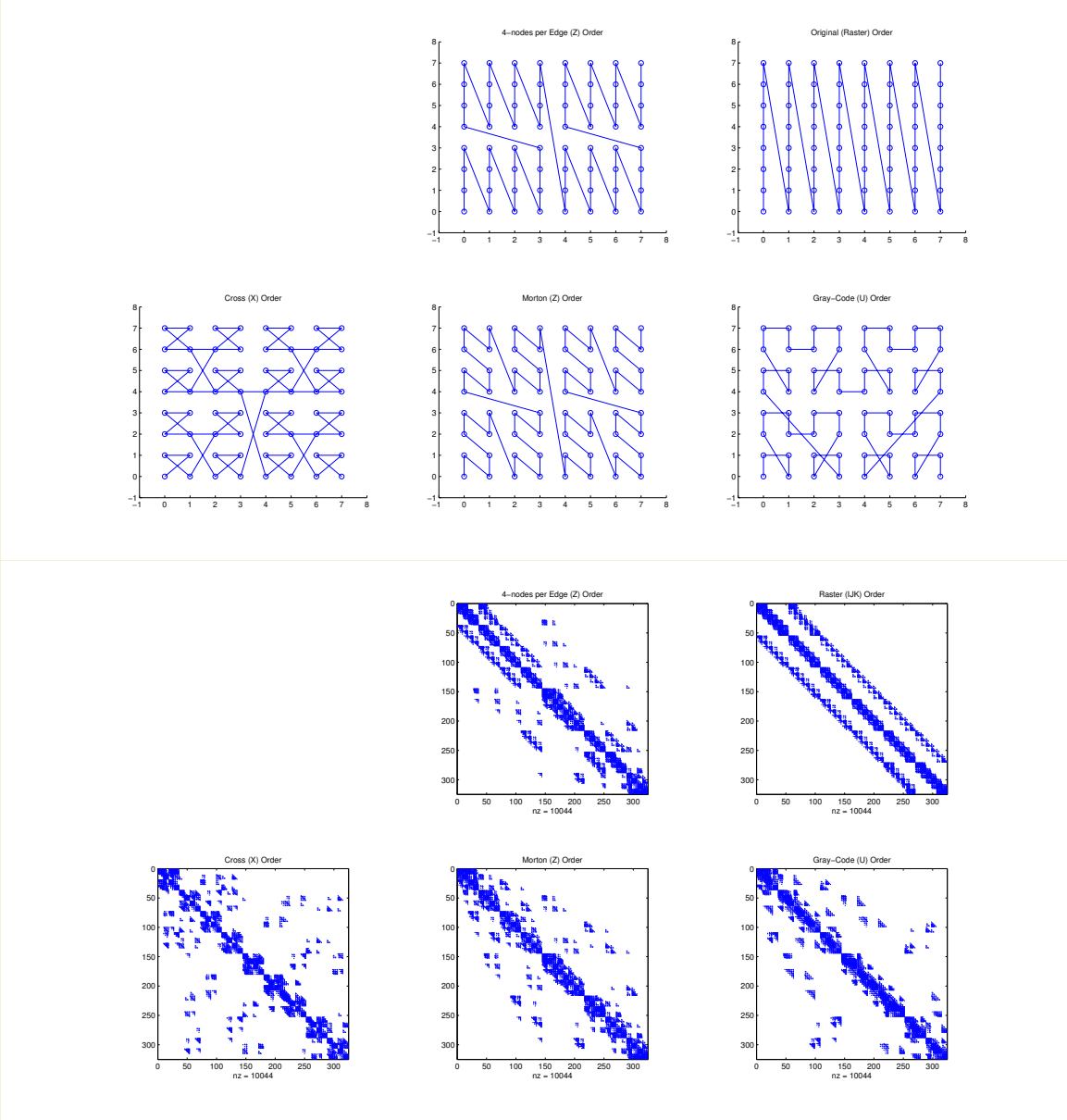


Figure 1.14: In order: a) node ordering test cases; b) original ordering of regular grid (raster); c) coarse grid overlay for hash functions ($hnx = 6$); d) example stencil ($n = 31$) spanning multiple Z's; e) spy of DM after orderings. Author's Note: TODO: Raster on top left. Add RCM ordering to top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

1.5 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.

BIBLIOGRAPHY

[1] Kdtreesearcher class. MATLAB R2013a Documentation (http://www.mathworks.com/help/stats/kdtreesearcherclass.html), Aug 2013. 1, 12

[2] Evan F. Bollig, Natasha Flyer, and Gordon Erlebacher. Solution to PDEs using radial basis function finite-differences (rbf-fd) on multiple GPUs. *Journal of Computational Physics*, 231(21):7133 – 7151, 2012. 13

[3] 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. 13

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

[5] 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

[6] 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. 1, 10

[7] 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. 1, 2, 4

[8] Natasha Flyer and Grady B. Wright. Transport schemes on a sphere using radial basis functions. *Journal of Computational Physics*, 226(1):1059 – 1084, 2007. 1

[9] Natasha Flyer and Grady B. Wright. A Radial Basis Function Method for the Shallow Water Equations on a Sphere. In *Proc. R. Soc. A*, volume 465, pages 1949–1976, December 2009. 1

[10] 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

[11] 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

[12] 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. 13

[13] S. Green. Cuda particles. NVidia Whitepaper, 2010. 2, 6, 7

[14] 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. 2

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

[16] Øystein E. Krog. GPU-based Real-Time Snow Avalanche Simulations. Master's thesis, Norwegian University of Science and Technology, June 2010. 2, 6, 7, 10, 11

[17] 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. 15

[18] 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

[19] 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. 12, 13

[20] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003. 11

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

[22] 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, 6, 8, 11, 12

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

[24] Andrea Tagliasacchi. kd-tree for matlab. http://www.mathworks.com/matlabcentral/fileexchange/21512-kd-tree-for-matlab, Sep 2010. 1, 2, 11, 12

[25] Andrea Tagliasacchi. kd-tree matlab. https://code.google.com/p/kdtree-matlab, Jun 2012. 4, 11, 12

[26] 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. 2, 6, 7, 11

[27] Holger Wendland. *Scattered Data Approximation*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2005. 1, 2, 6, 7, 11, 12

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

BIOGRAPHICAL SKETCH

Evan Bollig was born on the 6th of June, 1983. He grew up in the sleepy town of Sun Prairie, Wisconsin.

At a young age, Evan developed a sense of

Evan currently lives in Saint Paul, Minnesota.