Handout: Free Matrix Reordering for the Price of Fast(er) Stencil Generation

Evan F. Bollig
bollig@scs.fsu.edu
Florida State University

June 22, 2011

1 Introduction

A small group at FSU, namely Steven Henke, Gordon Erlebacher and Ian Johnson (apologies if I missed others) have been looking at algorithms and data-structures to perform fast nearest neighbor queries for implementations of Lagrangian methods like Smoothed Particle Hydrodynamics, and related methods in Molecular Dynamics that can be used for realistic fluid simulation or material fracture dynamics. Another application area is for flocking algorithms to simulate group movement as seen in schools of fish or herds of cattle.

One algorithm that caught their attention is called Locality Sensitive Hashing (LSH)[1], and as part of their code I have witnessed its ability to outperform k-D Trees when querying neighborhoods for large numbers of nodes in lower dimensions (1D, 2D, 3D). I am not confident of its ability in higher dimensions, although space-filling curves project multi-dimensional data down to one dimension, so its quite possible.

In another handout, I can compare the complexity of k-D Tree and Locality Sensitive Hashing. Suffice it to say that both building the data structure for LSH and querying neighbors can be more efficient in certain implementations[2] than using one of the most hardened k-D Trees available (see: [1]).

Why is this important? Within moving node systems (e.g., a particle systems), data structures must be updated at each iteration prior to querying neighborhoods, thus demanding particle methods to seek the most efficient methods to make this happen.

Luckily, RBF-FD is not a particle method, and it only requires neighbor queries once at initialization (to generate stencils and weights). However, that is a loaded statement since it might mislead one into believing its adequate to use their hardened and trusty k-D Tree for neighbor queries. Below we will find that using LSH with RBF-FD results in an additional compelling argument in its favor, aside from just faster stencil generation.

2 Algorithm

Here, we consider the two phases of the LSH algorithm implemented within my code. My implementation was written based on memory of discussion with Steve, Gordon and Ian, so it deviates from the true LSH algorithm in spots. This algorithm is not efficient, and I suggest referring to [1] for details on getting something more optimal.

First, preprocessing is required to build the structure:

1. imagine a $N_x \times N_y \times N_z$ cartesian grid overlay in your domain.
2. index each cell of the grid so that each has a unique index.
3. iterate through all nodes and generate a hash identifying the index of the cell that each node lies within

[1] amongst other names; we have always referred to it as "cell hashing" within our circles of discussion
[2] my implementation is not complete and not covered by this statement

4. append each node index to a list of indices for their cell
5. sort the list of cells (and hence nodes) according to the hash function.

Then, to query a node's k nearest neighbors:

1. select a node and generate its cell index (using the same hash as above)
2. append that cell's list of indices to a list of neighbors that will be queried
3. repeat above steps (1 and 2) for neighboring cells by generating cell hashes in contours propagating outward from the current cell[3]
4. break when maximum number of neighbors has been found[4].
5. compute distances from the center node to all neighbors in the query list
6. select the k nearest nodes based on distance[5].

3 Observations

One can choose many types of hash functions for preprocessing in LSH. In my code, I use a standard i, j, k indexing of the cells in the x, y, z directions respectively (with k varying fastest) because it required minimal thought to code. The cell hashing function is $[((i * N_y) + j) * N_z + k]$. Space filling curves like Z-indexing, Hilbert or Peano Curves can also be used and are preferred.

The last step of the Preprocessing stage calls for nodes to be sorted by their cell index—and potentially further by their location within the cell. In the case of randomly scattered nodes, and the i, j, k indexing, this step will group nearby nodes in a single dimension together in memory. Closer in memory implies a decreasing number of cache misses and free access to neighbor data. If the nodes are roughly pre-sorted (e.g., if working with a regular grid of nodes) this will have little to no effect. Note that a space filling hash function can further improve query time when working in 2D and 3D since they project multiple dimensional data into one dimension, maintaining locality of nodes in the higher space. The end effect is that a 3D stencil ends up sequential (or nearly sequential) in memory because nodes are nearby in the higher dimension space.

There are two options to sorting. First, an approximate sorting, where we sort by cell such that nodes remain ordered as they are in the cell, and cells are appended to a list of sorted nodes in order (easiest, costs nothing). Second, complete spatial sorting where individual cells are sorted before appending the cell to the list of sorted nodes (usually requires a space filling curve to sort nodes). We pick the first option in our tests below.

4 Problem

I have been using Centroidal Voronoi Tessellation (CVT) to distribute nodes in 2D and 3D geometries, combined with a k-D Tree implementation to generate my stencils. An example node distribution is shown in Figure 1.

One problem encountered has to do with memory access when stepping the PDE solution in time. Explicit time-stepping requires sparse matrix-vector multiply of the RBF-FD differentiation matrix and the current solution vector. Consider for example the matching Differentiation Matrix[6] for Figure 1 shown in Figure 2 when using stencils of size 13. Using a sparse matrix container, the rows of the matrix are condensed into tuples (row i, column j, value $w_{i,j}$) that are stored appropriately in memory; typically this implies separate

[3] I propagate in a square, but it would be better to do it like a bitmap circle. Best if we use a Z-ordering so propagation is reduced to simple bit operations.

[4] true LSH would break when a sufficient number of cells have been traversed radially to meet/exceed a requested neighborhood radius

[5] or select all within the requested radius

[6] We check the sparse matrix of laplacian weights since we're only concerned with non-zeros, and no boundary condition enforcement

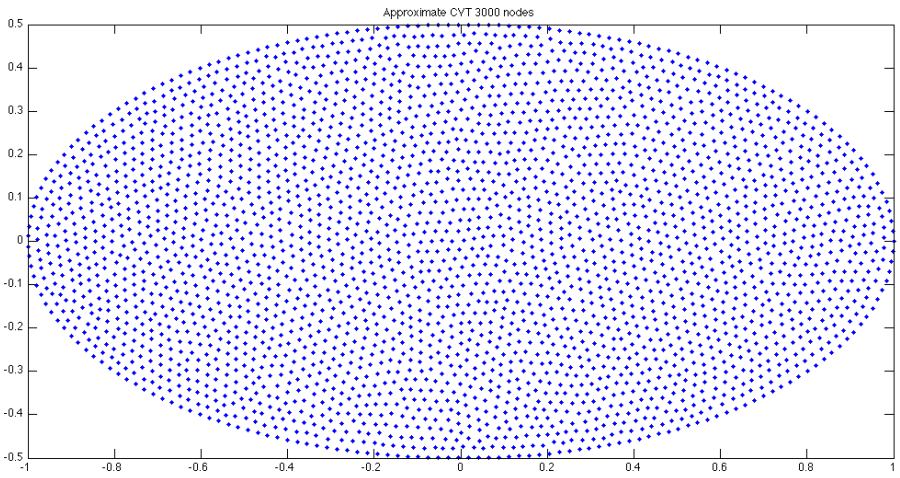

Figure 1: An approximately converged Centroidal Voronoi Tessellation of the ellipse.

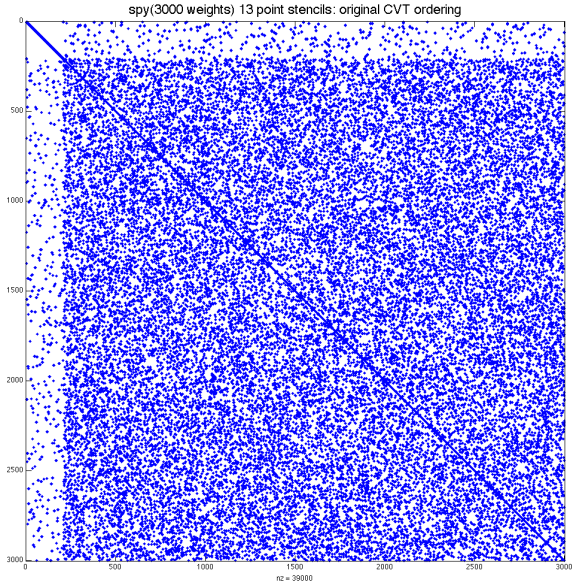

Figure 2: Sparsity pattern of RBF-FD Differentiation Matrix before stencil generation with sorting.

vectors for each element of the tuple. From Figure 2 it should be clear that even though a sparse matrix container can allow access to non-zero elements linearly in memory, the solution vector it is multiplied against will be sampled in a disjoint fashion.

On the GPU, random memory access like this can be a serious inefficiency. Even with caching available on modern GPU architectures, the non-zero entries of Figure 2 are too wide-spread to keep corresponding elements of the solution cached together.

Traditionally, sparse matrices like Figure 2 are re-ordered by algorithms such as Symmetric Reverse Cuthill-McKee (in Matlab: symrcm(A)) and Symmetric Approximate Minimum Degree Permutation (in Matlab: symamd(A)). Such methods demonstrate great success in condensing sparse matrices around the

diagonal. An example of the nicely condensed Symmetric Reverse Cuthill-McKee form for our differentiation matrix is shown in Figure 3.

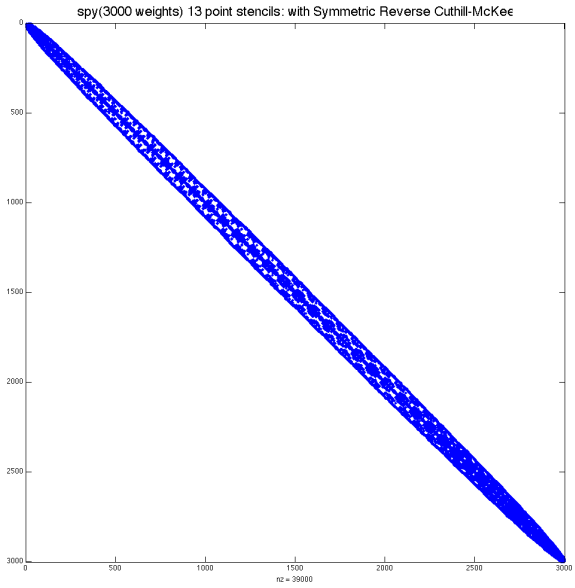

Figure 3: Sparsity pattern of RBF-FD Differentiation Matrix after permutation by Symmetric Reverse Cuthill-McKee (Matlab: symrcm).

5 What Falls Out of LSH

Since sorting nodes is part of the LSH preprocessing stage, let us consider what impact that has on our differentiation matrix.

In Figure 4 we use a 10×10 cell overlay and the i, j, k indexing on cells. We do not sort nodes within the cell, but order nodes grouped by cell index linearly in memory. In this Figure it should go without saying that the sparsity pattern has vastly improved, and block structures have started to appear that will allow efficient memory use.

The key concept here is: we hit two birds with one stone. Since we already needed to use either k-D Tree or LSH for stencil generation, without performing additional computation, LSH has significantly condensed our matrix and puts us one step closer to efficient memory access patterns while also pumping out stencils. Even without considering GPU computing, this is desirable. And its definitely not offered by k-D Tree.

6 Improving 2D Reordering

In Figures 5, 6, 7 we increase the cell overlay to 20×20, 30×30 and 40×40 respectively. Note that the higher resolution can be seen as subdividing cells within 10×10 overlay. Then, at higher resolution we are at least partially sorting nodes (spatially) almost in the way a Z-ordering would, but we're limited by the i, j, k indexing. Again, we observe an improvement in sparsity pattern for all three cases.

Interestingly, the sparsity pattern converges to a set of 3 bands. Also, the width of the non-zeros is inching toward a similar width as the Reverse Cuthill McKee.

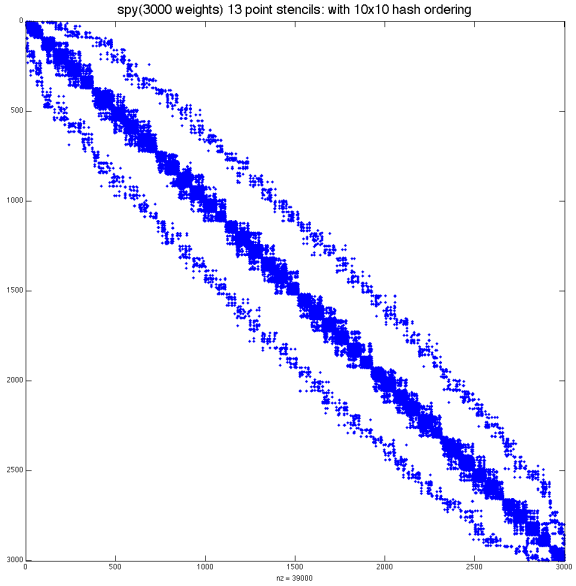

Figure 4: Sparsity pattern of RBF-FD Differentiation Matrix after stencil generation with a 10×10 cell overlay, and sorting nodes by cell (not within cell).

7 Considering 3D

A quick test was performed in 3D with the Ellipsoid shown in Figure 8 and stencil size 27. In Figure 9 we see the Symmetric Reverse Cuthill McKee result. Figures 10, and 11 show results from overlaying $40 \times 40 \times 40$ and $100 \times 100 \times 100$ cells respectively. The former shows 7 distinct "bands"; the latter 17[7]. Also, the overall width of each appears to be nearly the same as the Figure 9.

8 Moving Forward

- The next obvious step is to implement Morton- or Z-ordering to sort the nodes spatially and complete an implementation following [1]. Its possible with those orderings we would get a tighter packing of non-zeros near the diagonal. Also, with hierarchical space filling curves like Z-ordering we should be able to get much tighter packing with fewer cells, because nodes have all of their neighbors nearby in memory.
- It is not clear to what extent others have considered the impact of LSH in context of matrix re-ordering. Certainly, it improves structure, but how does it relate to algorithms like Reverse Cuthill-McKee and others? Different hashing functions will result in unique re-orderings; is one more optimal for the GPU (z-order, peano, hilbert, etc.)?
- Its not clear why those bands arise, and the significance of their numbering.

9 Conclusions

Using LSH with RBF-FD can both speed up neighbor queries, and improve the sparsity pattern of the differentiation matrix, which in turn aids in more efficient RBF-FD implementations. This is especially true

[7] its possible 100^3 is splitting the 7 bands into more because the overlay is placing so many empty cells between neighbors.

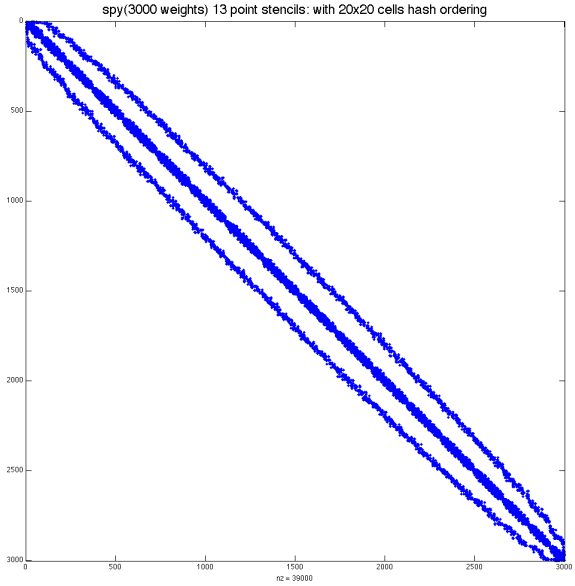

Figure 5: Sparsity pattern of RBF-FD Differentiation Matrix after stencil generation with a 20×20 cell overlay, and sorting nodes by cell (not within cell).

for the GPU[8].

References

[1] Connor, M., and Kumar, P. Fast construction of k-nearest neighbor graphs for point clouds. IEEE transactions on visualization and computer graphics 16, 4 (2009), 599–608.

[8]Did not have the time to quantify the impact on performance, but with everything sorted the probability is high that it is measurable

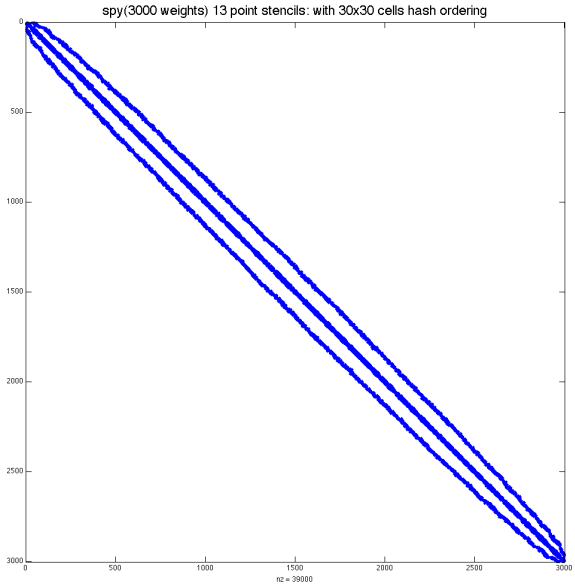

Figure 6: Sparsity pattern of RBF-FD Differentiation Matrix after stencil generation with a 30×30 cell overlay, and sorting nodes by cell (not within cell).

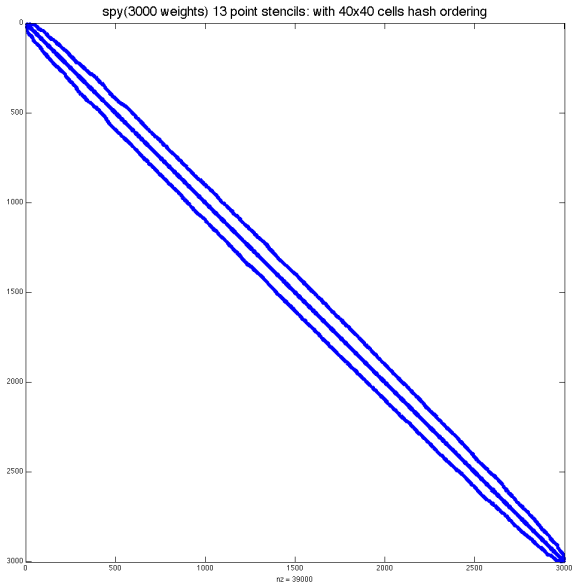

Figure 7: Sparsity pattern of RBF-FD Differentiation Matrix after stencil generation with a 40×40 cell overlay, and sorting nodes by cell (not within cell).

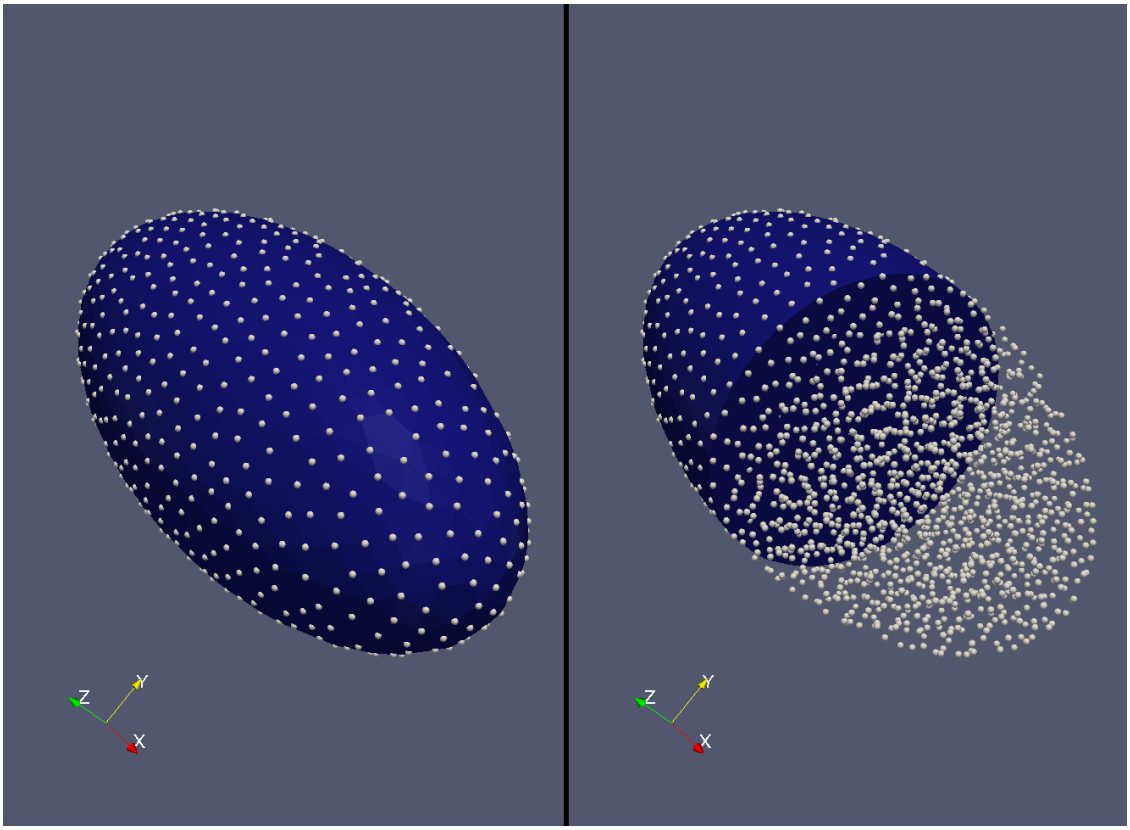

Figure 8: An approximately converged Centroidal Voronoi Tessellation of the ellipsoid with 3000 nodes. Left: node distribution on the surface. Right: nodes fill interior.

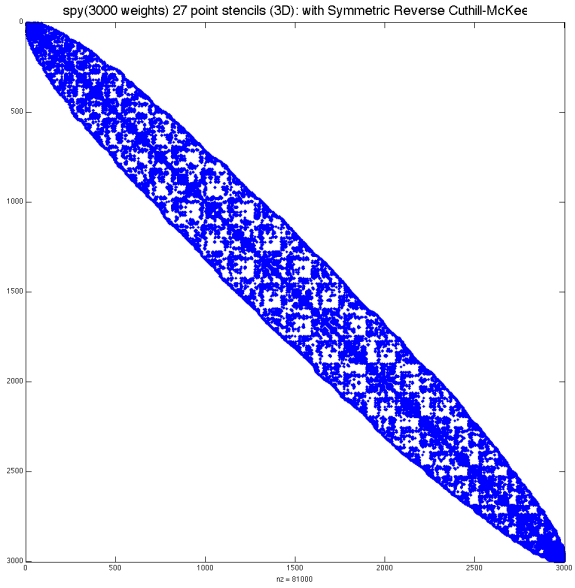

Figure 9: Sparsity pattern of RBF-FD Differentiation Matrix after permutation by Symmetric Reverse Cuthill-McKee (Matlab: symrcm).

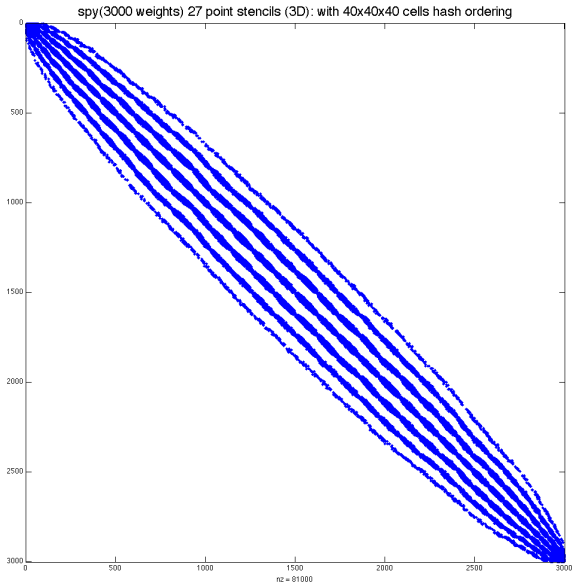

Figure 10: Sparsity pattern of the Ellipsoid Differentiation Matrix after stencil generation with a $40 \times 40 \times 40$ cell overlay, and sorting nodes by cell (not within cell).

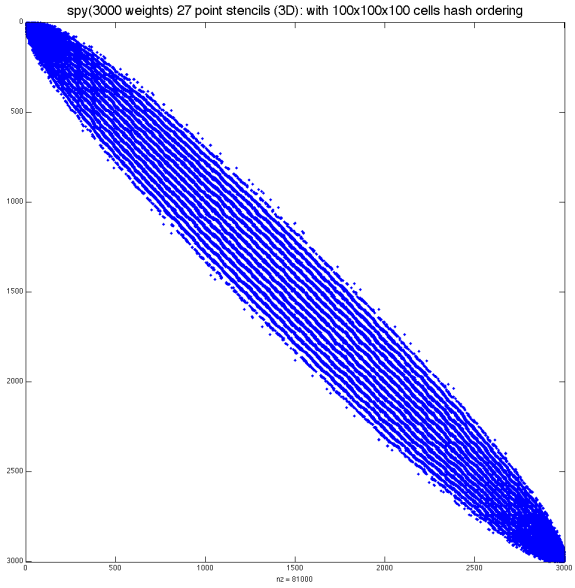

Figure 11: Sparsity pattern of the Ellipsoid Differentiation Matrix after stencil generation with a $100 \times 100 \times 100$ cell overlay, and sorting nodes by cell (not within cell).