Simple and Efficient Approximate Nearest Neighbor Search using Spatial Sorting

Marcelo de Gomensoro Malheiros
Center for Exact and Technological Sciences
UNIVATES
Lajeado, Brazil
Email: mgm@univates.br

Marcelo Walter
Institute of Informatics
UFRGS
Porto Alegre, Brazil
Email: marcelo.walter@inf.ufrgs.br

***Abstract*—Finding the nearest neighbors of a point is a highly used operation in many graphics applications. Recently, the neighborhood grid has been proposed as a new approach for this task, focused on low-dimensional spaces. In 2D, for instance, we would organize a set of points in a matrix in such a way that their x and y coordinates are at the same time sorted along rows and columns, respectively. Then, the problem of finding closest points reduces to only examining the nearby elements around a given element in the matrix. Based on this idea, we propose and evaluate novel spatial sorting strategies for the bidimensional case, providing significant performance and precision gains over previous works. We also experimentally analyze different scenarios, to establish the robustness of searching for nearest neighbors. The experiments show that for many dense point distributions, by using some of the devised algorithms, spatial sorting beats more complex and current techniques, like k-d trees and index sorting. Our main contribution is to show that spatial sorting, albeit a still scarcely researched topic, can be turned into a competitive approximate technique for the low-dimensional k-NN problem, still being simple to implement, memory efficient, robust on common cases, and highly parallelizable.**

***Keywords*-spatial sorting; k-nearest neighbors; parallel algorithms; data structures**

I. Introduction

The nearest neighbor search (NNS) is an essential operation in many scientific applications. More formally, given a set S of points in a d-dimensional space and a query point P, we want to find in S the closest point C to P, according to some distance metric. The more useful, and more general case, is to find a given number k of nearest neighbors (k-NN).

Our motivation is to run a massive 2D biological cell simulation, where cells can have different sizes and perform division at any time, as shown in Figure 1. Thus, both the NNS performance and the memory usage are our primary concerns. We also have the particular situation where the query point P is always within the search set S, and we need to efficiently gather its closest neighbors. This process is repeated for each point within the domain, at each simulation step.

After having reviewed several low-dimensional NNS techniques, we opted to evaluate the neighborhood grid approach, recently described in [1]. The basic idea has several advantages regarding our particular simulation problem when compared to traditional techniques like specialized k-d trees or variations of uniform grid arrangements.

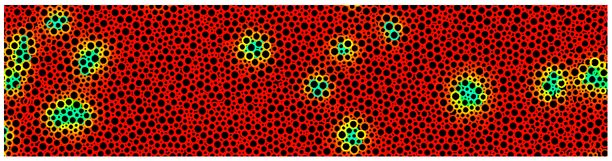
Fig. 1. Snapshot of a biological cell simulation.

In this paper, we present a significant improvement of the neighborhood grid applied to the 2D case, proposing new and more efficient spatial sorting strategies while experimentally measuring their performance and precision in distinct scenarios. We also discuss how several extensions can be made, in order to turn it into a competitive Approximate Nearest Neighbor technique. We show that this new technique is: **general**, behaving well for different point distributions; **memory efficient**, having very low data structure overhead; **fast**, by significantly reducing the number of comparisons needed to achieve the sorted state; and **dynamic**, adapting to a continually changing point set.

II. Related work

Li and Mukundan [2] presented a review on spatial partitioning methods with the focus on 2D crowd simulations, evaluating four data structures: grid, quadtree, k-d tree, and Bounding Interval Hierarchy (BIH). In their experiments, the grid performed better than the other three techniques. Recent k-d tree implementations make advances in GPU acceleration, overcoming limitations due to conditional computations and suboptimal memory accesses. Gieseke et al. [3] describes a buffered approach, organizing queries by spatial locality. Kofler et al. [4] use a specialized k-d tree to compute n-body simulations, built inside the GPU memory in distinct phases.

Fluid simulations using Smoothed Particle Hydrodynamics (SPH) methods are also dependent on k-NN, being typically performed on uniform grids. Green [5] presents a parallel linear sorting technique to group particles according to index similarity. Further improvements are presented by Ihmsen et al. [6], analyzing both spatial hashing and index sort. In another approach, Connor and Kumar [7] sorts the points along a linear sequence, following a Z-order, and placing them inside a matrix. After that the k neighbors of a point are found performing another local sort of $O(k \log k)$ complexity.

We note that approximate nearest neighbor search (ANNS) approaches were developed as an alternative to exact Voronoi diagrams. Of particular interest is the work of Har-Peled [8], which proposes a space decomposition that approximates a Voronoi diagram and has near linear size. Therefore it is possible to have a trade-off between accuracy and complexity.

Although quite efficient, these solutions still have shortcomings from our point of view, such as high memory overhead caused by the auxiliary data structures. Furthermore, as the set of cells from our biological simulation is continually growing, because new cells are created independently at different locations, we must update the data structure dynamically. So it is undesirable to reconstruct it at each time step. As a final demand, we also sought for an approach that is simple to parallelize on current GPUs.

III. Spatial Sorting

Although sorting is a classic topic in Computer Science, it is almost always devoted to a single sequence of elements, that is, a unidimensional list of comparable items. For clarity, we will call this *linear sorting*. For example, we can locate the $k = 2$ nearest neighbors for each real number of an array by performing two operations. First, we just sort the numbers in ascending order. Then, we can locate the nearest neighbors for a given array element i by just examining the elements with indices $i-2$, $i-1$, $i+1$, and $i+2$, as shown in Figure 2. Naturally, we need to adjust the search when dealing with elements near the array ends.

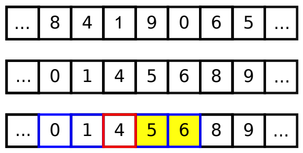

Fig. 2. Finding nearest neighbors: unsorted sequence (top), sorted array (middle) and searched elements (bottom). The query point is outlined in red, the candidates are in blue, and the two nearest neighbors are marked in yellow.

The term spatial sorting is sometimes used to name the process of ordering d-dimensional points along a space-filling curve so that the result is still a linear sequence. We may thus call it *linear sorting along a curve*. The more common schemes are using either the Hilbert curve or the Morton order curve (also known as Z-order) to establish a space traversal, where points are placed into discrete bins, which are then ordered when the curve is followed.

A better-suited approach of organizing points can be achieved by doing a bidimensional ordering on a given set of points, not restricted to a linear sequence, but arranged into a regular rectangular grid. If we employ a matrix as the underlying data structure, we can achieve better locality after the spatial sorting is performed. This also can enhance GPU memory access, as a 2D texture read operation can cache nearby queries coherently in two dimensions.

Surprisingly, there are very few references in the literature that cover the problem of organizing a matrix of points by both x and y coordinates simultaneously. In fact, the only references found are the ones that propose the main idea of the neighborhood grid [9], [10], [1]. However, it should be noted that the neighborhood grid approach described in previous works does not try to achieve a stable sorted state. Instead, only partial sorting is performed in each simulation step, mostly because of performance concerns, which leads to low precision when locating nearest neighbors. In this paper we explore ways to efficiently keep the matrix always fully sorted, which significantly improves the accuracy.

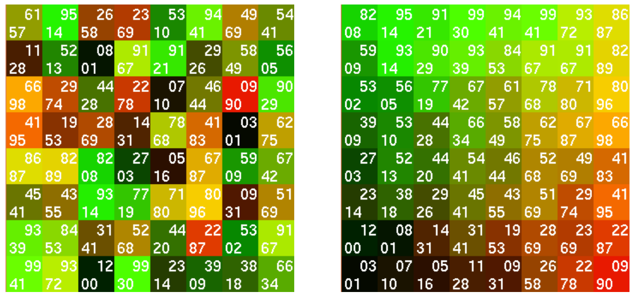

Fig. 3. Spatial sorting a matrix of random 2D points: before (left) and after (right).

We call *spatial sorting* the establishment of a stable ordering of 2D points into a matrix. An example of spatial sorting is shown in Figure 3, where x and y integer coordinates are shown on the bottom left and top right of each matrix element, respectively. For easy visualization the figures show color-coded points, where the normalized x and y coordinates are mapped to the red and green color channels, respectively. Also, to match the Cartesian plane, we opted to start the row numbering on the bottom of the matrix. Figure 4 depicts the candidate test, where nearby elements around the query point are tested to locate the two nearest neighbors.

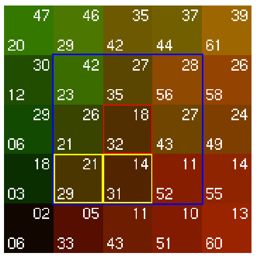

Fig. 4. Query point is shown in red, candidates in blue, and the two nearest neighbors in yellow.

We say that the matrix is spatially sorted when each element has its x coordinate greater or equal than its immediate left neighbor, and less or equal than its right neighbor. Also, the same condition holds for the y coordinate. It should be clear that achieving a sorted state is always possible, as shown in Figure 3. But it is not readily obvious that, from a given set of points, there are several possible sorted results, which are by definition stable. Thus, the problem is not to find *the* spatially sorted matrix, but one of the possible sorted states for it.

We have devised a simple algorithm to deterministically achieve a spatially sorted state, given a random set of n 2D points. For convenience, suppose $n = s \times s$, s being an integer. First, place all points into a continuous array of length n. Then, perform a standard linear sort on this array, comparing only the y coordinate, and thus creating an increasing sequence (regarding only y). After that, the array should be viewed as a row-major matrix, where the first s elements are in row 0. We now have that all points in row 0 must have y coordinates less or equal to the y coordinates of all points in row 1, and so on. Therefore, if now we perform a linear sort on each row, sorting by x coordinate, we will achieve the final, spatial ordering. Provided we employ an optimal linear sort, like merge sort, we have that the first step is $O(n\, log\, n)$, whereas the second step takes $s \times O(s\, log\, s)$, as we need to sort s rows, each one with length s. Combining the two steps and replacing s by $\sqrt{n}$, we find an overall time complexity of $O(n\, log\, n)$.

Note that the problem of sorting s unrelated lists of s real values has $O(n\, log\, n)$ as its established lower bound. We can build an associated spatial sorting problem by copying each unsorted list to a matrix row, setting its x coordinates. We can also define that for the i-th row, all its y coordinates will be set to i. If the matrix is then spatially sorted using the algorithm described earlier, the lists will be ordered. Therefore, $O(n\, log\, n)$ is also a lower bound for spatial sorting.

Although the previous algorithm (dubbed SIMPLE) reaches a sorted state, such arrangement provides consistently low precision when locating neighbors, as will be evaluated in Section V-A. This can be visually observed by comparing the color-coded sorted states reached by different sorting algorithms, as shown in Figure 5. That is the reason we explored iterative and more complex algorithms.

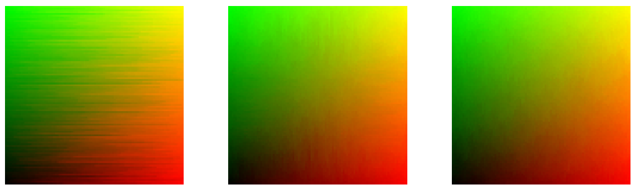
Fig. 5. Final sorted states for the same uniform point distribution, using distinct algorithms: SIMPLE (left), SOE (center), and SSI (right).

A general approach to reach a spatially sorted state is built on repeated linear sorts on the matrix, for several steps, as shown in Algorithm 1. For each step, we have two passes: a traditional linear sort is done on each row, and then another pass, where a linear sort is performed on each column of the matrix. After each step, the matrix will get more organized until a stable spatially sorted state is reached.

Algorithm 1 General approach to spatial sorting

```
$sorted \leftarrow false$
**while** $\neg sorted$ **do**
    **for** each row $r$ **do**
        run a linear sort on $r$, comparing $x$
    **for** each column $c$ **do**
        run a linear sort on $c$, comparing $y$
    **if** matrix is sorted **then**
        $sorted \leftarrow true$
```

A. Sorting strategies

Passos et al. [9] proposed a variation of the general approach, using as basis the odd-even linear sort algorithm. Instead of running the complete odd-even linear sort for each column and row, just one partial step was run. That is, for each row, a first pass would be done comparing (and swapping, if needed) adjacent pairs of values starting at odd matrix indices, and then a second pass comparing adjacent pairs starting at even indices. Then, two passes would be run for the columns. Therefore, a single step of the spatial odd-even sort would have four passes, as shown in Figure 6, where the $(0, 0)$ element is marked with a red dot.

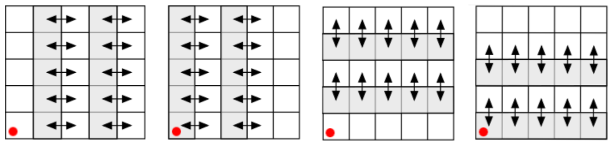
Fig. 6. Four passes for a single step of spatial odd-even sort, from [9].

Such odd-even step is then repeated until a stable configuration is achieved. The spatial odd-even sort (here named SOE) is simple to code and easily parallelizable, as the comparisons and exchanges within one pass can be all performed in parallel. However, this algorithm is also slow and memory-intensive, taking many steps to converge. In fact, the linear odd-even sort algorithm has quadratic time complexity.

It is easy to establish the time complexity for SOE. Given $n = s \times s$, one step will perform four passes on the matrix values. Each pass will need at most $s/2$ comparisons on each row or column, where a complete pass will perform $O(s^2)$ comparisons. A single step is still $O(s^2)$, and the number of steps will depend on the *sortedness* of the initial matrix. For the worst case, imagine that the point P with largest x and y values is at the bottom-left element. While sorting, the points with the larger coordinates move to the top and to the right, so the final element of P will be in the top-right corner. As a point moves at most one row or column at each step, it will take s steps for P to reach its final destination. Thus, the worst-case time complexity for SOE is $O(s^3)$ or $O(n^{1.5})$.

The linear odd-even sort is *adaptive*, that is, each step takes advantage of the previous partial ordering of data. Therefore, several runs of the spatial odd-even step will gradually sort the matrix. Such order is not achieved locally, like the recursion step of a linear merge sort, for example, but globally. As expected, just using efficient but non-adaptive linear sorting algorithms (like quicksort or bitonic sort) for repeatedly ordering the rows and columns alternately would still take too many comparisons, as observed in our experiments.

A better approach has to be adaptive, keeping the partial ordering from the last step and improving upon it as new steps are run. Also, we found that getting into a partially sorted state fast and then running another spatial sorting algorithm would converge much faster. Therefore, we employed a combination of linear algorithms that are known to be adaptive: insertion sort and Shell sort.

Linear insertion sort is known to have worst-case quadratic time complexity, but it is optimal when the input is already sorted, taking only $m - 1$ comparisons in an array of size m. This is especially relevant as the matrix gets progressively organized because some rows and columns will get fully sorted before others, thus not changing on further steps.

Linear Shell sort has time complexity that depends on the particular gap size employed. As it is based on linear insertion sort, it is also adaptive. But Shell sort has another desirable property: it has several passes, one for each gap size. When it runs with larger gaps, it is possible to jump elements over several positions within the linear sequence being ordered, which makes the data get more quickly into the sorted state. In fact, the last pass of Shell sort is always run with gap size 1, which means it is exactly the same as standard insertion sort. As the array is mostly ordered, this final pass is typically very efficient and ensures the final sequence is fully sorted.

Because linear Shell sort has several passes, we derived a spatial version, interleaving for each gap size g a parallel pass on each row, followed by another parallel pass on each column. A standard linear Shell sort pass of gap g is performed in each row or column. We observed best results when using just one step of the spatial Shell sort to create a "rough" ordering, which is then refined through iterative alternating linear insertion sort steps, as indicated by Algorithm 2.

Algorithm 2 Spatial Shell + insertion sort (SSI)

```
for each $gap$ in sequence $G$ do
    for each row $r$ do
        run a gapped insertion sort on $r$, comparing $x$
    for each column $c$ do
        run a gapped insertion sort on $c$, comparing $y$
$sorted \leftarrow false$
while $\neg sorted$ do
    for each row $r$ do
        run an insertion sort on $r$, comparing $x$
    for each column $c$ do
        run an insertion sort on $c$, comparing $y$
    if matrix is sorted then
        $sorted \leftarrow true$
```

This algorithm, dubbed SSI, arrived at a stable spatially sorted state faster than any other tested variation of the general approach (Algorithm 1). That is, it performed far better than repeatedly running a standard linear sorting algorithm (like quicksort) on each row and then on each column. For SSI, we have tested with the gap sequence suggested by [11], where $G = \{1750, 701, 301, 132, 57, 23, 10, 4, 1\}$. The particular complexity lower bound for this sequence has not yet been established in the literature, but we did run extensive experimental tests that confirm the efficiency of the SSI algorithm in the average case, against other strategies.

In order to standardize the experimental tests, we employed $1,000,000$ randomly generated points inside the $[0, 1] \times [0, 1]$ domain, which were then sorted by using different algorithms into a 1000×1000 matrix. For each sort, a fixed set of 100 distinct point distributions was used, measuring the CPU running time of the sorting process in each case. The running time was averaged, with the standard deviation shown as a red error bar in the figure (with the amplitude equal to 2σ).

The test programs were implemented in C++ on a Ubuntu 12.04 Linux system. The test machine is powered by an Intel Core i7-4500U CPU running at 1.80 GHz. Later on, parallel tests were done in the same configuration, but using an NVidia GeForce GT 750M GPU, with 384 CUDA cores, 2 Gb of GDDR5 RAM and with CUDA runtime 7.0.

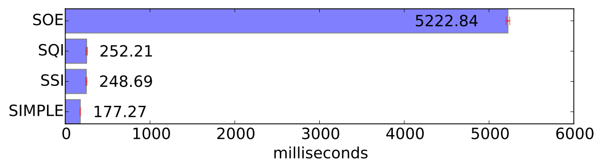

Fig. 7. Average running time for a million points, for a uniform point distribution.

Besides the SIMPLE, SOE and SSI algorithms, other variations were tested. Of interest, we will only mention SQI, which performs one row pass followed by one column pass of quicksort as the initial rough step, and then iterates until convergence with alternated passes of insertion sort. SQI performs marginally slower than SSI, but has the benefits of being simpler to implement, by reusing available quicksort routines. Furthermore, it is best fit for parallelization. As the matrix gets sorted, the number of performed swaps decays exponentially, where most of the last steps do very small changes to the matrix. Thus, for both SSI and SQI, we skipped ordering rows and columns that were not changed since the previous step, which gave a performance improvement.

To estimate the complexity of the iterated algorithms, we evaluated the mean number of comparisons for ten different uniform point distributions. This process was repeated for matrices with sizes ranging from 200×200 to 1200×1200. Then the resulting series were non-linearly fitted. We found as best fit a $n^{1.497}$ curve for SOE, which matches the previously estimated worst-case complexity of $O(n^{1.5})$. We also found a $n^{1.126}$ curve for SQI, and a $n^{1.088}$ curve for SSI, being n the total number of points. Although empirical, these last two curves hint at an average comparison complexity close to $O(n\, log\, n)$. Similar curves were also found for running time.

B. *Point distributions*

We have performed experimental tests in many synthetic 2D point distributions, to assess how well the spatial sorting and neighbor gathering algorithms performed in typical and not-so-common cases. We have evaluated 12 different distributions.

The first set of distributions is shown in Figure 8, which contains both continuous arrangements over the domain $[0, 1] \times [0, 1]$. The **uniform** distribution generates random points with x and y coordinates within the interval $[0, 1]$. The **gradient** distribution uniformly varies the density of points along the $x = y$

diagonal. The **gaussian** distribution generates coordinates with mean 0.5 and standard deviation 0.2. The **shuffle** distribution arranges points into a regular grid, and then randomly shuffles them inside the matrix.

Figure 9 exemplifies some discrete distributions. The **inside** distribution selects points uniformly scattered inside a circle centered on $(0.5, 0.5)$ with radius 0.5. The **clusters** distribution generates five random and non-overlapping circles within the domain, evenly picking points that lie inside one of these circles, whereas the **holes** distribution picks points outside all those circles. The **streets** distribution places all points evenly distributed inside overlapping axis-aligned rectangles.

Another set of distributions is shown in Figure 10, which illustrates some problematic cases for nearest neighbor queries. The **compact** distribution places all points evenly distributed in one fifth of the normalized domain, with y coordinates limited to the interval $[0, 0.2]$. The **triangle** distribution generates points in half of the domain. The **tilted** distribution applies random rotations to an arrangement identical to *streets* and finally, the **diagonal** distribution places points arranged into a thin rectangle tilted by 45 degrees.

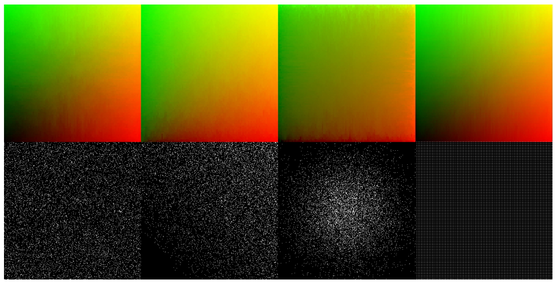
Fig. 8. Color-coded spatially sorted matrices (top row) for some continuous point distributions (bottom row), using SSI. The distributions are, from left to right, respectively: uniform, gradient, gaussian, and shuffle.

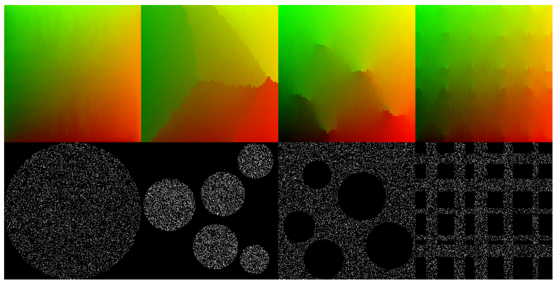
Fig. 9. Color-coded spatially sorted matrices (top row) for some discrete point distributions (bottom row), using SSI. The distributions are, from left to right, respectively: inside, clusters, holes, and streets.

Figure 11 shows the average running time for sorting several distributions with algorithms SQI (shown in blue) and SSI (in orange), using 100 different random seeds each. The SOE algorithm is not pictured as it performed in average 20 times slower than the other two. As we can see, most of the distributions have a similar sorting performance, with continuous distributions being faster to sort spatially. The exception is the **diagonal** distribution, which was found to be the pathological case for the SSI algorithm.

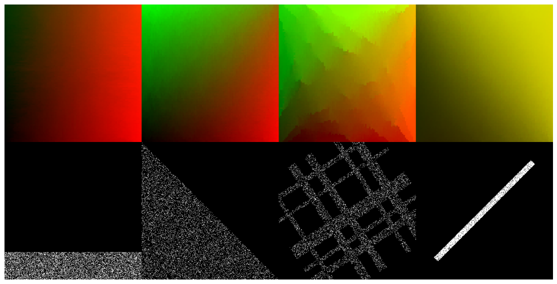
Fig. 10. Color-coded matrices (top row) for some unusual point distributions (bottom row), using SSI. The distributions are, from left to right, respectively: compact, triangle, tilted, and diagonal.

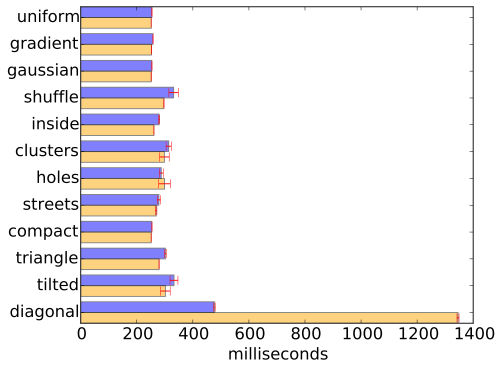

Fig. 11. Average running time for different distributions: SQI (blue) and SSI (orange).

C. Updating the matrix

If the points stored in the matrix change, like in a typical simulation iteration, it is straightforward to return to the sorted state by simply running a spatial sorting algorithm again. Depending on the magnitude d of changes, it may be cheaper to run a few steps of alternating insertion sort (dubbed SI), as parts of the matrix could still be in ordered state. Table I lists the update times for the SI and SSI algorithms, given the number of points changed and their change intensity.

TABLE I
UPDATE TIME (MS) FOR TWO SPATIAL SORTING ALGORITHMS, GIVEN PERCENT OF POINTS CHANGED AND MAGNITUDE OF CHANGE.

	SI			SSI		
	0.01	0.1	0.2	0.01	0.1	0.2
1%	32.98	52.63	64.71	65.72	91.32	103.08
2%	37.78	65.71	82.38	70.66	104.64	123.78
10%	57.55	108.38	137.34	88.95	149.63	170.15
25%	74.39	143.68	193.37	107.24	174.35	190.83
50%	92.89	176.07	237.81	121.98	189.90	206.36

IV. GENERAL DATA STRUCTURE

This section provides a brief discussion of the issues that have to be handled when using spatial sorting as part of a general data structure. Such operations are needed to support the ongoing research into a massive biological cell simulation, as we strive to keep the memory overhead at a minimum and also handle the dynamic creation of cells by division. Previous works ([9], [10], [1]) only cover the situation where the matrix is filled with a constant number of points.

The typical matrix arrangement is a square, but its aspect should match the point distribution. We thus need the matrix dimensions to reflect an adequate rectangle placed on the plane. The general approach is to find the ratio of the number of columns and rows that better approximates the ratio of x and y intervals in the domain. Otherwise, the precision when searching for neighbors would be negatively affected.

It should be clear by now that all points must lie in a domain which is topologically equivalent to a square. The case of points on the entire surface of a sphere, for example, cannot be handled by the current approach. In this case, even if the point coordinates are stored as a pair of angles, the proximity of points on the poles will not be correctly identified.

On most situations, it is not possible to exactly fit all points into a rectangular matrix. Instead, at least an extra row or column will be needed, with some elements left empty. Empty elements are easy to handle, as they need to store the maximum floating point value for both the x and y coordinates. They are then compared with other points as usual by the spatial sorting algorithms, which will make them accumulate on both the topmost row and the rightmost column. When testing candidates, such empty positions should be ignored.

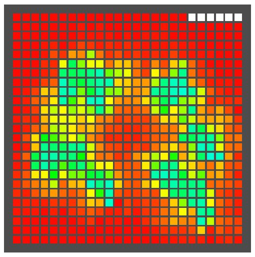
Fig. 12. Matrix with empty elements, shown in white.

We may allocate all matrix elements sequentially in memory, either in row-major or column-major order. A size extension would increase the row or column count, filling the newly "uncovered" positions (at the end of allocated memory) with empty elements, as in Figure 12. After that, a full spatial sort is needed, otherwise all points would get shifted along the matrix by the size change. This scheme provides the minimum memory usage.

After we created some empty elements inside the matrix, there are several ways in which a new point can be inserted. The simpler approach is just to select one of the available empty elements and set its x and y coordinates to the new point to be added. A good choice would be the leftmost empty element or the bottommost one, as the new point will already be with lower coordinate values than the empty one, which will "slide" to the correct matrix element during the next sort.

Point removal is straightforward. We should mark as empty the associated position, and after a sorting update is run, such element will move to the top or to the right of the matrix.

Point location happens when the query point is not already associated to a element inside the matrix, which is not the recommended usage for this approach. Still, such operation may be necessary, and therefore we need to employ a search scheme that can locate the rough placement of the query point, and then perform a local search using a Moore neighborhood.

As we only have independent ordering of rows and columns, we cannot use a spatial bisecting algorithm like a k-d tree. We also cannot walk on the matrix towards a minimum distance regarding the query point (q_x, q_y), as the relative sorting creates several local minima. Thus, we need to perform a more extensive search to locate the center for a "good" candidate neighborhood. We have experimented with a procedure comprised of three steps. First, for each row of the matrix, we perform a binary search on the x coordinate, to locate the element closer to q_x. From the elements selected in each row, we pick the point C that has the closest y coordinate to q_y. As this is not enough to accurately pinpoint the nearest neighbor for all situations, we finalize by analyzing candidates inside a Moore neighborhood centered at C. It can be shown that this algorithm has time complexity $O(\sqrt{n}\ \log\ n)$.

V. Results

In this section, we evaluate the sorting algorithms proposed using both CPU and GPU implementations. We first access the performance and precision of the neighbor gathering phase. Then we compare the neighborhood matrix to state-of-the-art k-d tree and index sort implementations.

A. *Neighbor gathering*

Once a full sort is established, we can select candidates to test for nearest neighbors in the vicinity of a given matrix element. We then perform neighbor gathering, where the query point is already in the dataset. For a typical simulation application, this phase tends to be more computationally expensive than the sorting phase, because we need to test all candidates and then perform the actual calculation between near points, like collision of particles or interaction between agents. Therefore, the neighborhood size and shape must be chosen carefully. Several strategies are possible, and traditionally a Moore neighborhood is used to collect candidate points. The choice of a small neighborhood may not find all the correct nearest points on the plane, but would be more efficient because fewer tests would be needed. On the other hand, a large neighborhood could adversely impact on the computing time as many candidates could turn into points not near enough. In [1] is suggested an adaptive approach to collect all required points, but in our work we opted to use the fixed square Moore-like neighborhoods with $m = \{8, 24, 48, 80, 120\}$ neighbors.

We have measured the running time of the neighbor gathering phase in the typical situation that, for each point in the dataset, all k nearest neighbors are collected. Then we compare the found set of nearest points to a "ground truth": an exact k-d tree algorithm run on the same set of points. This comparison makes possible to identify precisely the number of misses for each combination of spatial sorting and candidate neighborhood size. It is also helpful to compare the total time taken, comprised of the setup time (spatial sorting or k-d tree construction) added with the query time (neighbor gathering or k-d tree point query).

We have also measured the precision of other approaches, to compare their effectiveness against spatial sorting. We tested the approximate k-d tree search from CGAL, which takes an *epsilon* parameter specifying the error margin to be applied when searching for nearest neighbors (we used 1.0 as it achieves a similar precision to SOE). We also tested against a partial sort, as done in [1], where a single spatial quicksort step is run before the gathering phase.

For the following results the matrix size is again 1000×1000, with 10 different runs for each algorithm, and unless noted, with the **uniform** point distribution. In Figure 13 we list the comparison of setup and query time, for a reference exact k-d tree search and several other algorithms, when just one nearest neighbor is located for an m-48 neighborhood. Precision is shown as a percent of points that had a wrong nearest neighbor in comparison with the exact results. We see that although the setup time for the k-d tree is very small, most of the time is spent on the tree traversal to locate nearest points. On the other hand, for the SQI and SSI spatial sorts the setup time is longer and the candidate search is very efficient, thus reducing the overall time spent. Even though an approximate search is also employed for the k-d tree, the performance gains are small. Also shown are the results for the SOE and SIMPLE algorithms, which have lower precision than SQI and SSI. Finally, it is clear that just a partial sorting pass is not enough to guarantee a useful precision; even with an m-120 neighborhood the miss rate would be 90.29%.

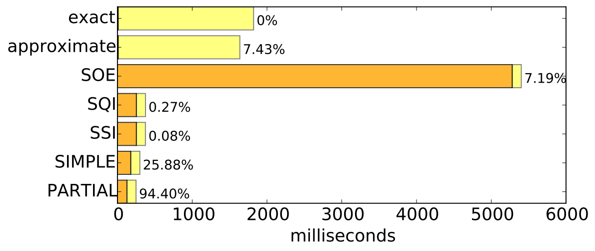

Fig. 13. Setup time (orange), query time (yellow) and precision for several algorithms, with $k = 1$ and search neighborhood m-48.

TABLE II
TOTAL TIME (MS) AND PRECISION FOR SEVERAL SPATIAL SORTING ALGORITHMS AND SEARCH NEIGHBORHOODS, WITH $k = 1$.

	SOE		SQI		SSI	
type	time	miss%	time	miss%	time	miss%
m-8	5305	25.2408	278	15.5910	277	14.8582
m-24	5340	11.9231	313	1.6832	314	1.2017
m-48	5401	7.1885	374	0.2652	374	0.0782
m-80	5478	4.5036	450	0.0867	454	0.0051
m-120	5563	2.8363	536	0.0442	546	0.0003

A comparison between spatial sorting algorithms is shown in Table II, again for a uniform distribution. As noted before, the algorithms generate quite different final sorted results, which directly affect the search precision. The SSI algorithm is consistently the one that gives the lower miss rate. In Figure 14 a brief comparison is made of the running time for SSI and precision when using an m-48 neighborhood, against different point distributions. We can see that the optimal cases are the continuous distributions, even though most other discrete situations have a miss rate around 2.5%. The only exception is again the pathological **diagonal** case.

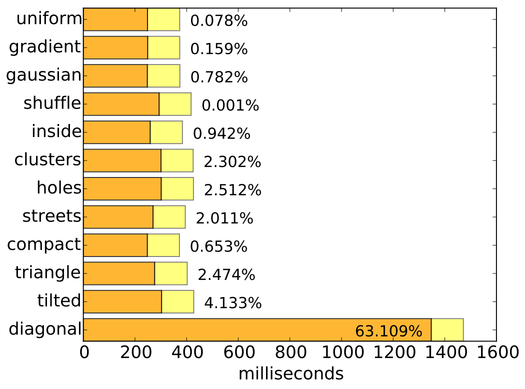

Fig. 14. Setup time (orange), query time (yellow) and precision for several distributions, with m-48 neighborhood and $k = 1$.

TABLE III
TOTAL TIME (MS) AND PRECISION, WHEN VARYING k.

	k = 4		k = 8		k = 16	
type	time	miss%	time	miss%	time	miss%
m-8	367	75.4719	400	99.8042	439	100.0000
m-24	521	11.7654	622	42.0174	741	94.9046
m-48	685	0.8329	848	4.6112	1061	28.9598
m-80	871	0.0492	1097	0.2938	1423	2.9639
m-120	1060	0.0032	1342	0.0173	1756	0.1991

In Table III, we briefly summarize results for varying k while using the SSI on a uniform distribution with different candidate neighborhoods. A *hit* is achieved only when there is an exact match between the set points found. Thus, the precision measure is very strict, based on the exact k-d tree from CGAL. That said, even for a *miss*, most of the nearest points are correctly located. For example, for $k = 16$ and with a m-180 search neighborhood, for all 10^6 points, only 0.1991% of them got a wrong set of neighbors. However, taking into account all 10^6k neighbors located, only 0.0141% were different from the correct ones.

B. Comparison with other NNS algorithms

We have evaluated our proposed spatial sorting algorithms against current NNS strategies, in both serial (single-threaded CPU) and parallel (CUDA) configurations[1]. We briefly describe the results, showing that for dense and rather homogeneous distribution of points, spatial sorting can outperform both k-d tree and index sorting.

We have built a 2048×2048 set of uniformly distributed 2D points and averaged the setup time (to build the data structure), the query time (to locate $k = 10$ nearest neighbors for all given points), and the miss rate (strict, that is, considering

[1]Sample code is available at: http://github.com/mgmalheiros/spatial-sorting

TABLE IV

AVERAGE SETUP TIME, QUERY TIME, TOTAL TIME, AND MISS RATE FOR A SET OF 2^{22} UNIFORMLY DISTRIBUTED POINTS INSIDE A SQUARE DOMAIN, WITH $k = 10$.

implementation	setup (s)	query (s)	total (s)	miss%
CGAL k-d tree	0.040	19.300	19.340	0
nanoflann k-d tree	1.598	10.974	12.572	0
SSI m-120	1.212	5.021	6.233	0.0336
SSI m-80	1.212	3.836	5.048	0.5807
SSI m-48	1.212	2.666	3.878	8.3461
CUDA m-120	0.739	0.677	1.418	0.5018
CUDA index sort	0.045	1.322	1.367	0
CUDA m-80	0.739	0.483	1.224	1.6144
CUDA m-48	0.739	0.314	1.053	10.6954

the full set of neighbors). The test hardware is the same as described previously. We also computed the total time, which includes both the setup and query time. The results are shown in Table IV, ordered by average total time.

For the serial case, we tested against the Nanoflann k-d tree, which is a highly-optimized C++ template for low-dimensional NNS. In fact, it was the fastest 2D implementation we have found. As a reference, we also kept the times given by the CGAL k-d tree, as it can scale to higher dimensions.

The parallel implementation of spatial sorting used two steps of bitonic sort, followed by several steps of spatial insertion sort (SI). As spatial Shell sort cannot be efficiently run in parallel because of its many alternating passes in x and y directions, we opted to employ bitonic sort as a standard sort. Thus, bitonic sort would be run in each row in parallel, and then on each column, again in parallel.

As a reference parallel NNS, we used a finely-tuned implementation of index sort for a uniform grid, based on the CUDA *particles* demo, which is described in [5]. In this case, we adapted the source to provide a 2D NNS, using on a 256×256 grid of buckets. The timings are also shown in Table IV.

Although the setup time is high for spatial sorting, it starts from an unordered matrix of points. Therefore, we can use spatial sorting to update the matrix after positions change, which can give significant gains for the repeated iterations of a simulation. For example, given a change of magnitude 0.001 on 10% of the points distributed in a unit square $[0, 1] \times [0, 1]$, the update sort would take only 0.231 s on average, against 0.739 s for the CUDA implementations, which would get a significant edge against index sort, as the total time would drop from 1.418 s to 0.908 s (for the m-120 case). By using bitonic sort, we are in fact reaching a different sorted state than SSI, which is equivalent to using quicksort for each row and column, which reflects on the slight higher miss rates for the parallel implementation. Finally, it should be noted that index sort demands two complete matrices in the GPU memory, to improve memory locality when performing the query phase.

VI. CONCLUSION

We have shown that the use of spatial sorting algorithms can provide an efficient yet simple to implement technique for 2D approximate nearest neighbor search. In particular, neighbor gathering in dense arrangements of points seems to be the best scenario.

We can highlight our major contributions as the proposal of new spatial sorting algorithms and the establishment of a lower bound for its time complexity. We discussed how to construct a general data structure, describing how to dynamically handle the insertion and removal of points, besides the point location problem. We then have evaluated the performance and accuracy of this approach in many different and representative scenarios. We also showed that the overall results are competitive with current NNS algorithms like k-d tree or index sort based on uniform grids.

Although we did not address 3D spatial sorting in this paper, its extension and usefulness is straightforward, having achieved a precision similar to the 2D case on other tests we made. However, it remains to be evaluated in higher-dimensional situations.

As future work, we plan to assess possible optimizations to the GPU implementation. Another venue for research is the design of an actual spatial sorting network which could be optimal in parallel architectures. We also would like to explore this approach applied to other traditional NNS problems in Computer Graphics, like global illumination. And there is also need to evaluate how to map it to other topologies (like points on the surface of a sphere).

ACKNOWLEDGMENT

We wish to thank Tadeu Zubaran and Marcus Ritt for very helpful discussions about the lower bound for spatial sorting.

REFERENCES

[1] M. Joselli, J. R. da S. Junior, E. W. Clua, A. Montenegro, M. Lage, and P. Pagliosa, "Neighborhood grid: A novel data structure for fluids animation with gpu computing," *Journal of Parallel and Distributed Computing*, vol. 75, no. 0, pp. 20 – 28, 2015.

[2] B. Li and R. Mukundan, "A comparative analysis of spatial partitioning methods for large-scale, real-time crowd simulation," in *Proc. 21st Intl. Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2013, pp. 104–111.

[3] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel, "Buffer k-d trees: Processing massive nearest neighbor queries on gpus," in *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, 2014, pp. 172–180.

[4] K. Kofler, D. Steinhauser, B. Cosenza, I. Grasso, S. Schindler, and T. Fahringer, "Kd-tree based n-body simulations with volume-mass heuristic on the gpu," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 2014, pp. 1256–1265.

[5] S. Green, "Particle simulation using cuda," *NVIDIA Whitepaper*, 2010.

[6] M. Ihmsen, N. Akinci, M. Becker, and M. Teschner, "A parallel sph implementation on multi-core cpus," *Computer Graphics Forum*, vol. 30, no. 1, pp. 99–112, 2011.

[7] M. Connor and P. Kumar, "Parallel construction of k-nearest neighbor graphs for point clouds," in *Proceedings of the Fifth Eurographics/IEEE VGTC conference on Point-Based Graphics*, 2008, pp. 25–31.

[8] S. Har-Peled, "A replacement for voronoi diagrams of near linear size," in *focs*. IEEE, 2001, p. 94.

[9] E. Passos, M. Joselli, M. Zamith, J. Rocha, A. Montenegro, E. Clua, A. Conci, and B. Feijó, "Supermassive crowd simulation on gpu based on emergent behavior," in *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, 2008, pp. 81–86.

[10] M. Joselli, E. B. Passos, M. Zamith, E. Clua, A. Montenegro, and B. Feijó, "A neighborhood grid data structure for massive 3d crowd simulation on gpu," in *Games and Digital Entertainment (SBGAMES), 2009 VIII Brazilian Symposium on*. IEEE, 2009, pp. 121–131.

[11] M. Ciura, "Best increments for the average case of shellsort," in *Fundamentals of Computation Theory*. Springer, 2001, pp. 106–117.