COMPARISON OF SPATIAL DATA STRUCTURES IN OPENMP–PARALLELIZED STEERING

Alexander Wirz, Claudia Leopold, Björn Knafla
University of Kassel, Research Group Programming Languages / Methodologies
Wilhelmshöher Allee 73, 34121 Kassel, Germany
wirz@student.uni-kassel.de, {leopold,bknafla}@uni-kassel.de

KEYWORDS

Parallelization of Simulation; Partitioning, Mapping, and Scheduling

ABSTRACT

With the invent of multicore CPUs, parallelization becomes an important issue in the programming of computer games. We consider steering, i.e., the coordination of a large number of agents to simulate swarm-like behavior. Steering and in particular its neighborhood search component are compute-intensive and profit from parallelization. In this paper, we combine OpenMP threads with the use of spatial data structures, considering three structures: k-d tree, grid, and cell array with binary search. In addition to parallelization, we tuned the performance by e.g. optimizing cache usage. The focus of the paper is an experimental comparison of the three structures, based on a parallel version of the OpenSteer library. In nearest-neighbor search and updates, we found none of the structures to be superior to the others, and provide a detailed account of their pros and cons.

1. INTRODUCTION

As parallel architectures are becoming commonplace, novel application areas arise. One of them is computer games in which, among others, artificial intelligence components can profit from faster hardware. In this paper, we consider steering, a technique that models the movement of computer-controlled characters (agents) as a combination of simple steering behaviors. Examples include obstacle avoidance and alignment to the movement direction of neighbors. From individual behaviors, a seemingly intelligent group behavior emerges. Steering is deployed in games such as Fable and Alarm for Cobra 11.

We carried out experiments with OpenSteer (Reynolds, 1999, 2004), an open-source C++ library that implements steering behaviors, and its associated testbed OpenSteerDemo that supports rapid prototyping of combinations of the behaviors. Two scenarios were considered: 1) a group of pedestrians following a path and thereby avoiding collisions, and 2) a swarm of birds moving according to rules of separation, alignment and cohesion. OpenSteerDemo has been parallelized with OpenMP threads in previous work (Knafla and Leopold, 2007), and we refer to this parallel version.

As found by profiling, the clearly most compute-intensive component of steering is neighborhood search. Therein, each agent computes the c nearest from among those agents who are within a certain radius around the agent. A straightforward algorithm considers all pairs of agents and requires time $O(n^2)$ for n agents.

Nearest-neighbor search can be speeded up by using spatial data structures, which are well-known from areas such as database management systems and computer graphics, where they are used for various types of geometric queries. In general, these structures organize data according to their coordinates, and thus store neighbors closer to each other. We use the structures to store agent data, which allows us to restrict nearest-neighbor search to only a subset of agents.

Several types of spatial data structures have been suggested in the literature and can be classified into flat and hierachical structures (Velho et al., 2002). This paper investigates k-d tree, grid, and cell array with binary search, of which the k-d tree is a hierarchical and the others are flat data structures. We consider two variants of the grid, based on hashing and modulo functions, respectively. A potentially infinite world is assumed for all structures, i.e., coordinates are not bounded.

Our application runs a sequence of simulation steps. In each step, it searches for the nearest neighbors of all agents, and updates the states of all agents. As agents move, states and in particular coordinates change dynamically. A specific requirement of our application is parallel nearest neighbor queries and parallel updates, i.e. multiple agents issue queries at the same time. Data structures that support these operations have been implemented with C++. Performance was tuned to make each structure as efficient as possible. For instance, we grouped queries from close agents to improve cache usage.

Our main contribution is an experimental comparison of the performance of the three structures. Briefly stated, there was no clear winner. Although the flat data structures clearly outperformed the k-d tree, they require careful application-specific tuning. The k-d tree, in contrast, is more robust towards changes in the application setting. Of the flat structures, the cell array with binary search is on advantage if agent density is high, and inferior other-

Proceedings of the 2008 High Performance
Computing & Simulation Conference ©ECMS
Waleed W. Smari (Ed.)
ISBN: 978-0-9553018-7-2 / ISBN: 978-0-9553018-6-5 (CD)

wise. Only of the two grid variants, the modulo grid was a clear winner.

Sect. 2 of this paper starts with background on steering, OpenSteer, and OpenSteerDemo, including OpenMP parallelization. Then, Sect. 3 introduces the three data structures. We explain the update and nearest-neighbor search operations, including parallel implementation. Moreover, Sect. 3 comprises an analytical comparison among the structures. Sect 4 is devoted to experiments, covering experimental setting, results, and discussion. Finally, Sect. 5 concludes the paper.

2. STEERING AND OPENSTEER

Steering is an artificial intelligence technique that is used in the implementation of computer games. It can be deployed to simulate swarms, flocks and other groups of agents, for which a complex group behavior emerges from simple individual behaviors of the group members. These individual behaviors are composed of elementary behaviors that may include (Schnellhammer and Feilkas, 2004):

- *separation*: keep some minimum distance from neighbors to avoid collisions,
- *cohesion*: stay near the center of neighbors to form a group, and
- *alignment*: adjust direction and velocity of movement with neighbors, to provide for a coordinated action.

Each elementary behavior is represented by a steering vector that is computed by vector operations from the agent's own state (position, orientation, velocity), the states of its neighbors, and possibly further information on the local environment (e.g. obstacles). The steering vectors of different behaviors are combined, e.g. by weighted summation, to compute the agent's new state.

OpenSteer and OpenSteerDemo can be used to develop and tune steering behaviors. In particular, OpenSteerDemo allows a user to write plugins for certain simulation scenarios. This paper refers to two scenarios that we adopted from Reynolds (Reynolds, 1999, 2004):

- *Pedestrian:* Here, agents (pedestrians) walk through a two-dimensional world, following the path in Fig. 1 from *a* to *g* and backwards, repeatedly. To avoid collisions, each agent observes its environment and computes steering vectors according to pathfollowing, agent- and obstacle-collision avoidance rules. Note that only nearest neighbors (as described below) are taken into account in these computations. Thus, steering avoids but does not preclude collisions, the remaining collisions are taken care of by other components of a game.
- *Boids:* This scenario assumes a three-dimensional world. Boids stands for birds-like objects or bird-oids, i.e., the plugin simulates a swarm of birds who move according to the separation, cohesion, and alignment rules described above. As in the pedestrian plugin, the computation of steering vectors is based on nearest neighbors only.

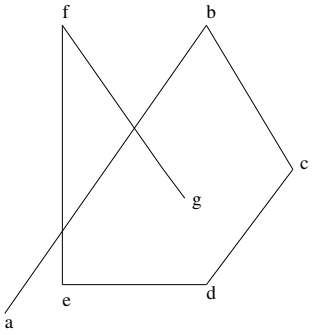


Figure 1: Path for pedestrian plugin

The term nearest neighbor needs further explanation. In both scenarios, it is assumed that an agent can only see its local environment, which is modeled by a fixed-size circle (for pedestrian), or a fixed-size sphere (for boids) around the agent, respectively. An agent must fulfill two requirements to be a nearest neighbor of another agent: 1) it must be situated within maximum distance d from that agent, and 2) it must belong to the c closest neighbors of that agent. Obviously, the nearest-neighbor relation is not symmetric.

OpenSteerDemo runs a main loop as it is typical for games. Each step of the iteration simulates one discrete time step, and is subdivided into an update stage and a graphics stage. The former logically moves all agents, and the latter renders the new state of the game world. We use the term frame rate to denote the number of main loop steps that can be carried out per second. Thus, frame rate is a measure for the speed of the application, and a high frame rate corresponds to a low running time.

In previous work by the same authors, the application has been refactored and parallelized (Knafla and Leopold, 2007). The parallel version deploys a user-configurable number of OpenMP threads, each of which simulates multiple agents. In the parallel version, the update stage has been split up into a simulation sub-stage and a modification sub-stage, which are separated by barriers. The simulation sub-stage starts with a nearest-neighbor search for all agents, and then computes and combines steering vectors based on the states of neighbors. During the modification sub-stage, agents update the world state by writing their new individual states to a shared data structure. Since the simulation sub-stage comprises read accesses only, and the writes of the modification sub-stage refer to disjoint data, each of the two stages can internally be run in parallel, without any need for synchronization.

In the base version of OpenSteerDemo (Knafla and Leopold, 2007), the running time of each step is dominated by the time for nearest-neighbor search. This version deploys a straightforward search algorithm that we

denote as *brute-force* in the following. It assumes agents, or more specifically the data that represent the agents' states, to be stored in a list, and deploys a doubly-nested loop. The outer loop runs through all agents to invoke nearest-neighbor search, and the inner loop runs through all agents to select the c closest neighbors.

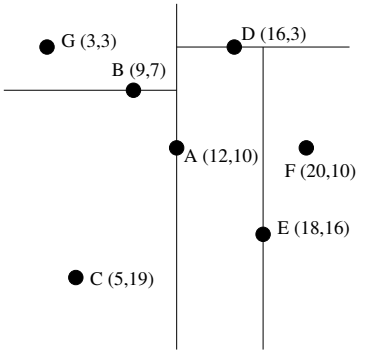
Obviously, in the inner loop, it would be sufficient to consider agents that are reasonably close. This can be achieved with spatial data structures that allow to identify close agents without touching the others. The following section describes three structures that we experimented with in this paper: k-d tree, grid, and cell array with binary search.

3. PARALLEL DATA STRUCTURES AND IMPLEMENTATION

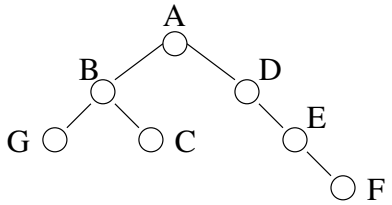
k-d tree

The k-d tree (Bentley, 1975) is a special type of binary search tree for data with k-dimensional keys. In our case, keys represent agent positions, and $k = 2$ (for pedestrian) or $k = 3$ (for boids). We refer to the homogeneous variant, in which data are held by both internal nodes and leaves. As in any search tree, a node's key is larger than the keys of all nodes in its left subtree, and less than or equal to the keys in its right subtree. The special feature of k-d trees is the definition of this less-than relation. Depending on node level, it discriminates by one of the k coordinates: first coordinate for the root, second coordinate for nodes at level one, and so on, cyclically.

Figure 2 gives an example for $k = 2$. Here, subtrees of `A` are discriminated by x-coordinate, subtrees of `B` and `D` by y-coordinate, and subtrees of `G`, `C` and `E`, if existent, by x-coordinate, again.



(a) Points in a two-dimensional space



(b) The corresponding k-d tree

Figure 2: Example for k-d tree

A k-d tree may be balanced or not, and hence its height may vary from O(log n) to O(n), where n denotes the number of nodes. Operations *search* and *insert* are straightforward: As in standard binary search trees, they start from the root and recursively branch into the left or right subtree, depending on a comparison between the given node's key and that of the root. Of course, the comparison must be based on the respective coordinate, and thus the level needs to be taken care of. As in standard search trees, a newly inserted node becomes a leaf.

For *delete*, let us denote the node to be deleted by p, and assume its position to be known from a previous search. If p is a leaf, delete is trivial. Otherwise, without loss of generality, let p discriminate its subtrees by x. Then, the contents of p is replaced by the contents of q, the node with smallest x-coordinate from among the nodes in p's right subtree (if there are multiple such nodes, any can be taken). In case of an empty right subtree, the left and right subtrees are exchanged before selecting q. After having replaced the contents of p by that of q, the algorithm deletes q by applying the same scheme recursively.

The algorithm for *nearest-neighbor search* takes as input a center point, which is the position of an agent, and the radius of the search region, which is a circle or sphere around it. Starting with the root, it checks for each node on its way through the tree whether it is inside the search region. If it is, the corresponding agent is a candidate for nearest neighbor, and the algorithm compares it to the others found so far to decide whether it is among the c closest. If the agent is outside the search region, the value of its discriminator coordinate is compared to the maximum and minimum values that this coordinate may take for the search region, and the search proceeds with either the left, right, or both subtrees, accordingly.

Parallel invocation of nearest-neighbor search for multiple agents does not require any changes to the data structure, as all accesses are reads. We allocate frequently used data structures such as lists of nearest neighbors once per thread instead of once per nearest neighbor search, to avoid expensive system calls from a parallel region.

After each simulation step, the positions of all agents need to be updated. Unfortunately, k-d trees are ill-suited for this operation since they have a static structure, i.e., movement requires rearrangement of the tree. Of course, any update can be realized by a delete followed by an insert, but these operations are expensive and nontrivial to parallelize (JáJá, 1992). Therefore, we decided to rebuild the tree after each simulation step, which has the additional advantage that the generated tree is balanced. Experiments showed this variant to be faster than inserts/deletes.

Grid

A grid subdivides the world into disjoint cells (Samet, 2006). We refer to equal-sized cells that correspond to rectangles and cuboids, respectively. Grid cells are indexed with k-dimensional vectors, where k denotes the

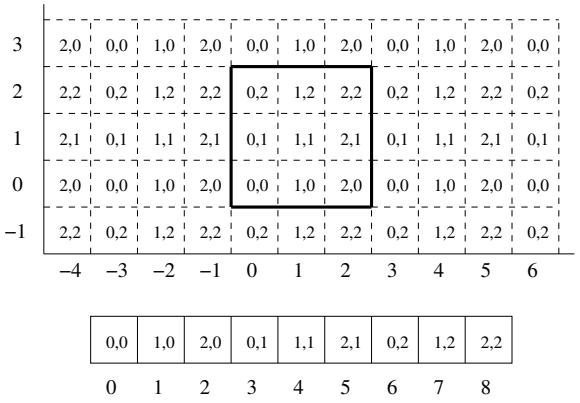


Figure 3: Assignment from grid cells to buckets with modulo function. The selected region is marked in bold. There is a one-to-one correspondence between cells of the region and buckets, as illustrated in the lower part of the figure.

dimensionality of the world. From an agent's coordinates, the index of the grid cell holding the agent is computed by integer divison through cell side lengths. If cells have extent $10 \times 10 \times 5$, for instance, an agent at coordinates $(23,7,17)$ belongs to grid cell $(2,0,3)$.

From the assumption of an unbounded world, there is an infinite number of grid cells. The grid data structure groups them into a finite number m of buckets and stores them in an array. Each element holds a list (implemented by an STL `vector`) of those agents that belong to a grid cell assigned to the bucket.

Note that the assignment of agents to buckets comprises two steps. First, agents are assigned to grid cells by dividing their coordinates by the cell side lengths, and then grid cells are assigned to buckets by another function. For the second step, we considered two options: hashing and modulo. In both cases, the bucket index is computed from an agent's coordinates within constant time. For hashing, we used function

$$\text{hash}(x,y,z) = (x \cdot p_1 \ \text{ xor } \ y \cdot p_2 \ \text{ xor } \ z \cdot p_3) \ \% \ m$$

where (x,y,z) is the index of the grid cell, and $p_1 = 73856093$, $p_2 = 19349663$ and $p_3 = 83492791$ are primes. The function has been taken from Teschner et al. (Teschner et al., 2003) to balance the number of grid cells per bucket.

For the modulo function, a core grid is spanned on a finite region of the world. Core cells have a one-to-one correspondence to buckets. Cells outside core are mapped to core by a modulo operation. Figure 3 illustrates the assignment. For example, grid cells $(1,2)$ and $(-2,-1)$ are mapped to bucket 7.

In both variants of the grid, *search* is realized by first computing the agent's bucket, and then searching this bucket linearly. *Insert* and *delete* start selecting the bucket, as well. Then, insert appends the agent to the corresponding list, whereas delete first locates the agent and then replaces it by the last entry.

Nearest-neighbor search selects the set of grid cells that intersect with the search region by simple geometric calculations, and runs through all buckets that contain at least one grid cell of interest. In both the hashing and modulo variants, the algorithm traverses each bucket only once if the search region overlaps with multiple cells of the same bucket. In the hashing variant, a bitset with one entry per bucket is used for bookkeeping. The modulo variant does not consider individual cells, but directly computes the buckets from agent positions.

For performance tuning, we modified the parallel version of OpenSteer, which has an outermost `for` loop that runs through all agents, splits this loop into equal-sized chunks, and maps each chunk to a thread. In the tuned version, the outermost loop runs through buckets, and an equal-sized chunk of buckets is assigned to each thread. Threads process buckets one after another. Since each bucket holds a group of grid cells, and agents from the same grid cell are close, nearest neighbor queries issued for agents from the same bucket have search regions with much overlap. Thus, when processing these queries, a similar set of buckets needs to be considered, which can likely be kept in cache. The improvement in locality comes at the price of worse load balancing, as buckets differ in their number of agents. Nevertheless, in experiments we observed a performance increase by about 35%.

Updates are simple writes of coordinates as long as an agent stays within the same cell. If it crosses cell boundaries, it must be deleted from the old cell, and inserted into the new one. As with nearest-neighbor search, each thread is responsible for multiple buckets. To deal with boundary crossings, it maintains a private list of agents moving away. At the end of the update phase, all private lists are processed sequentially. It would have been possible to parallelize this step as well, but not worthwhile as only 0.5 to 2 % of agents are affected.

Cell Array with Binary Search

Like a grid, the cell array with binary search subdivides the world into disjoint cells and assigns them to buckets (Mauch, 2003). Unlike a grid, it includes only $k-1$ dimensions into the partitioning, but uses the last coordinate to sort entries of each bucket.

Figure 4 depicts a two-dimensional world divided into cells of side lengths 10. As in the modulo variant of grids, cells of a selected region (here 0 to 30 on the x-axis) directly correspond to buckets. To assign an agent to a bucket, first the index of the cell holding the agent is calculated by dividing the agent's coordinates by the cell side lengths, and then the cell is mapped to the selected region by the modulo operation. Within each bucket, agents are sorted by their kth coordinate.

Search is accomplished by first selecting the bucket, and then carrying out a binary search. For *insert* and *delete*, data have to be moved to create or fill a gap, which is less efficient than in grids.

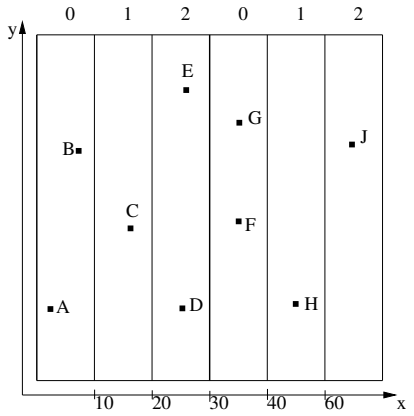


Figure 4: Cell array with binary search

For *nearest-neighbor search*, as in grids, the algorithm selects all buckets that intersect with the search region, but chooses buckets based on coordinates of only $k-1$ dimensions. When searching a particular bucket, it makes use of the sorted order. As illustrated in Fig. 5, the search region defines minimum and maximum coordinates in the kth dimension for agents of interest ($y_{\min}$ and $y_{\max}$). In each bucket considered, the algorithm carries out a binary search to locate $y_{\min}$, and then investigates consecutive agents until their kth coordinate exceeds $y_{\max}$.

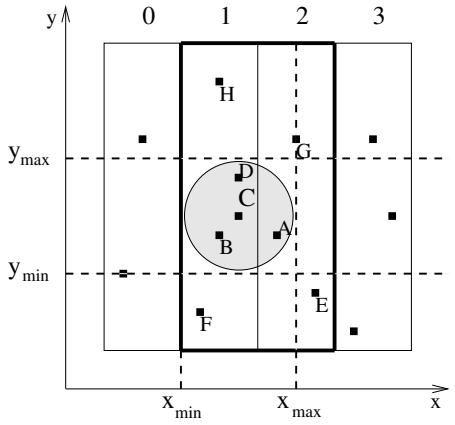


Figure 5: Nearest-neighbor search in a two-dimensional cell array.

Updates are realized as in grids, except that buckets are unsorted after modifications. Thus the grid algorithm is followed by a parallel sorting step in which each thread sorts multiple buckets. Despite our data being presorted, the STL sorting algorithm turned out to be faster than insertion sort, and was therefore deployed.

Analytical Comparison

Table 1 analytically compares the data structures, refering to the operations of interest in OpenSteerDemo. Formulas assume that the operations are applied to all agents. The table uses the following abbreviations:

- n: number of agents
- m: number of buckets
- f: average number of agents in the search region
- g: average number of grid cells in the search region
- h: average number of agents in grid cells that intersect with the search region

	Buildup	**Nearest-Neighbor**	**Update**
Brute-F.	$O(n)$	$O(n^2)$	$O(n^2)$
k-d Tree	$O(n \cdot \log n)$	$O(n(\log n + f))$	$O(n \cdot \log n)$
Grid	$O(m+n)$	$O(n(g+h))$	$O(n)$
Cell A.	$O(m+n^2)$	$O(n(g \cdot \log(n/m) + h))$	$O(n \cdot \log n)$

Table 1: Average running times of operations buildup, nearest-neighbor search, and update for different data structures

Formulas have been taken from Samet (Samet, 2006) and Mauch (Mauch, 2003). The table shows that all spatial data structures outperform the brute-force algorithm. Differences between the structures are minor. Note that part of the formulas can not be directly compared as, for instance, h differs between grid and cell array even for identical scenarios. The formulas refer to a sequential implementation. As we will see in Sect. 4, the running time of our application is dominated by nearest-neighbor search. For this operation, analytical speedup with p threads is linear for all structures, since threads process different agents independently.

4. EXPERIMENTS

Both the pedestrian and boids scenarios were run with 1000 agents, selecting $c = 7$ neighbors. We set the search radius to 12 units in the pedestrian plugin, and to 9 units in the boids plugin. In the boids scenario, we bounded the world by a sphere of radius 50 units, with a bird leaving the sphere immediately re-entering it at the diametrically opposite point. Agent positions were initialized by randomly placing pedestrians onto the path, and randomly placing birds in a sphere of radius 20 units, respectively.

We run experiments for 7200 simulation steps of OpenSteerDemo and report averages over three such runs. OpenSteerDemo allows to separately measure running times for different stages, and we made use of this feature. Experiments were carried out on a dual-processor dual-core AMD Opteron 270 machine with 2GB memory. OpenSteer / OpenSteerDemo was compiled by the Intel compiler version 10.0.23 with options `-std=c99 -O2 -inline-level=2 -openmp -openmp-report1 -fp-model fast -xW`.

	1 Thread	2 Threads	4 Threads
Brute-Force	66,94	34,54	18,39
k-d tree	59,22	31,39	16,30
Grid (Hashing)	41,38	20,97	11,94
Grid (Modulo)	28,48	14,75	7,67
cell array	24,10	13,32	6,77

Table 2: Running time of nearest-neighbor search for pedestrian plugin (in seconds).

	1 Thread	2 Threads	4 Threads
Brute-Force	40,60	21,72	11,71
k-d Tree	24,73	13,91	7,63
Grid (Hashing)	15,66	8,53	4,89
Grid (Modulo)	10,62	6,02	3,40
Cell Array	10,93	5,84	3,32

Table 3: Running time of nearest-neighbor search for boids plugin (in seconds).

Tables 2 and 3 list running times for nearest-neighbor queries. Note that values are for 7200 simulation steps with 1000 queries each.

The values in tables 2 and 3 refer to tuned variants of the flat data structures, in which we selected side lengths of cells and number of buckets as depicted in Table 4. Note that our data structures can handle an unbounded world, but in both scenarios the world is limited. Therefore, we decided for a one-to-one correspondence between buckets and grid cells for the modulo grid. For the hashing grid, such a one-to-one correspondence could not be achieved.

	Side length of cells		Number of buckets	
	Pedestrian	Boids	Pedestrian	Boids
Grid (H.)	15	18	80	500
Grid (M.)	15	15	343	343
Cell Array	10	10	100	100

Table 4: Selected parameters after performance tuning

A comparison of tables 2 and 3 reveals that nearest-neighbor search always takes longer in the pedestrian than in the boids scenario. This difference is partly due to the larger search radius, for which more nodes or buckets need to be investigated. It can even be observed for the brute-force algorithm, because of the need to select the c closest from among the neighbors in the search region. Moreover, the boids scenario more uniformly distributes agents in space, such that less birds than pedestrians are candidate for nearest neighbor.

Both tables clearly demonstrate that spatial data structures speed up nearest-neighbor search. In both cases,

- flat data structures outperform the k-d tree
- the modulo variant of the grid outperforms the hashing variant
- the cell array with binary search is slightly faster than the modulo grid.

Differences between the data structures are partly due to differences in the number of agents that need to be looked at during the search. This presumption was verified experimentally by counting agents (Wirz, 2008). One reason the hashing grid is inferior to the modulo grid is the need to search all grid cells assigned to a bucket, in contrast to the one-to-one correspondence in the modulo grid.

The number of agents to be looked at is about the same for k-d tree and modulo grid. Nevertheless, the modulo grid is faster, which is due to the realization of buckets by STL vectors, in contrast to the pointer-based structure of the k-d tree.

In the cell array, only about half the number of neighbors as in the modulo grid need to be looked at. The performance gain is less, though, since the cell array requires an additional operation: locating $y_{\min}$ by binary search. Whether or not the expense for this operation pays depends on the number of agents in a bucket.

As mentioned above, cell side lengths have been tuned for the flat structures, since we observed a major impact on running times. If side lengths are small, many cells and the corresponding buckets need to be considered; if cells are large, selected cells may hold many agents outside the search region. Thus, optimal values depend on both agent density and search radius. Experiments proved our hypothesis that a one-to-one correspondence between cells and buckets performs best in a bounded world.

Tables 2 and 3 show that parallelization of nearest-neighbor search was successful for all data structures, confirming the analysis in Sect. 3. Speedups range between 3.1 and 3.7 with 4 threads for the different data structures and scenarios.

Parallelization of the updates was less successful, as tables 5 and 6 show. We observed speedups of 1.3 to 2.2 with 4 threads for the flat data structures (Wirz, 2008), and slowdowns for the k-d tree. For the k-d tree, rebuilding the tree was about twice as fast as deleting and reinserting the nodes. As tables 5 and 6 show, the cell array performed best for the updates, as well. The hashing grid is slightly faster than the modulo grid, and the k-d tree is slowest.

The low speedups may be due to not having spent much time on tuning updates, since their impact on the overall running time is low. The fraction of running time spent in different phases of the application is illustrated in Fig. 6. It shows that spatial data structures significantly speed up the simulation phase (sum of search, update and simulation sections) to the point that further efforts are only worthwhile for the graphics phase. Results for the pedestrian scenario are similar and have been omitted for brevity.

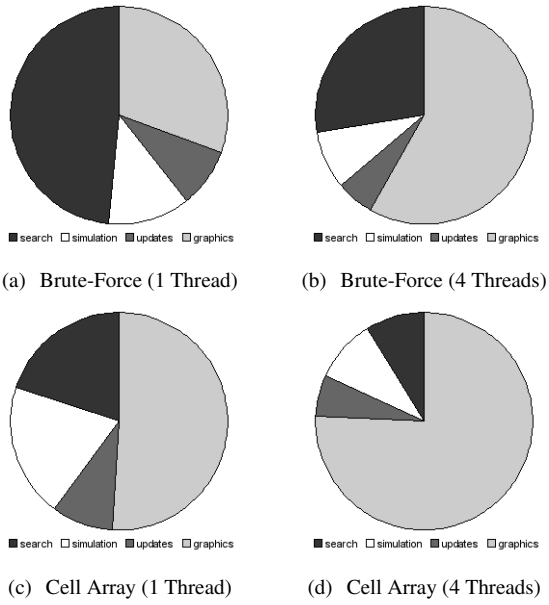
In our application, the speedups can be used to either increase the frame rate, or to handle more agents. Table 7 shows how many agents can be handled at a frame

	1 Thread	2 Threads	4 Threads
k-d Tree	3,19	4,40	4,52
Grid (Hashing)	1,51	1,06	0,70
Grid (Modulo)	1,42	1,47	1,09
Cell Array	1,13	0,96	0,69

Table 5: Running time of update for pedestrian plugin (in seconds)

	1 Thread	2 Threads	4 Threads
k-d Tree	2,24	3,20	3,38
Grid (Hashing)	1,52	1,16	0,78
Grid (Modulo)	1,40	1,09	0,97
Cell Array	1,08	0,95	0,72

Table 6: Running time of update for boids plugin (in seconds)



(a) Brute-Force (1 Thread) (b) Brute-Force (4 Threads)

(c) Cell Array (1 Thread) (d) Cell Array (4 Threads)

Figure 6: Fraction of running time spent in different phases for boids plugin

rate of 30 frames per second with the various data structures. Increases are significant and demonstrate the success of parallelization in combination with spatial data structures.

Finally comparing all data structures, the flat structures perform much better than the k-d tree, but have the drawback of requiring tuning. In particular, they can not deal with frequent changes in search radius. The k-d tree is robust towards such changes, and still performs better than the brute-force algorithm. Of the flat structures, the cell array appears somewhat faster than grids, but this outcome depends on scenario. The cell array is on advantage if cells are densely populated, but for sparsely populated cells grids may be the better choice. Of the two grid variants, the modulo grid was the clear winner.

	Brute-F.	k-d Tree	Grid (M.)	Cell A.
Ped.	2850	3100	4400	4700
Boids	3300	5000	7800	8200

Table 7: Maximum number of agents that can be simulated at 30 frames per second with 4 threads

5. CONCLUSIONS

The paper has shown that use of spatial data structures can significantly speed up steering in both sequential and parallel settings. Experiments were based on OpenSteer / OpenSteerDemo, and considered three structures: k-d tree, grid, and cell array with binary search. There was no clear winner, although a certain advantage for the cell array could be observed.

Future work should consider more data structures, especially more hierarchical structures such as range trees (Samet, 2006), other scenarios, and data structures for a bounded world.

REFERENCES

Bentley, J. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517.

JáJá, J. (1992). *An Introduction to Parallel Algorithms*. Addison-Wesley.

Knafla, B. and Leopold, C. (2007). Parallelizing a Real-Time Steering Simulation for Computer Games with OpenMP. In *Proc. Parallel Computing (ParCo)*, pages 219–226.

Mauch, S. (2003). *Efficent Algorithms for Solving Static Hamilton-Jacobi Equations*. PhD thesis, California Institute of Technology.

Reynolds, C. W. (1999). Steering Behaviors For Autonomous Characters. In *Proc. Game Developer Conference*, pages 763–782.

Reynolds, C. W. (2004). OpenSteer Website. `http://opensteer.sourceforge.net`.

Samet, H. (2006). *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann.

Schnellhammer, C. and Feilkas, T. (2004). Steering behaviors. `http://www.steeringbehaviors.de`.

Teschner, M. et al. (2003). Optimized spatial hashing for collision detection of deformable objects. In *Proc. Vision, Modeling, Visualization*, pages 47–54.

Velho, L., Gomes, J., and Figueiredo, L. H. (2002). *Implicit Objects in Computer Graphics*. Springer.

Wirz, A. (2008). Parallele Räumliche 3D Datenstrukturen für Nachbarschafts-Abfragen. Master's thesis, Universität Kassel.