A Packed Memory Array to Keep Moving Particles Sorted

Marie Durand, Bruno Raffin, François Faure

Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS (2012)
J. Bender, A. Kuijper, D. W. Fellner, and É. Guérin (Editors)

A Packed Memory Array to Keep Moving Particles Sorted

Marie Durand[1,2,3], Bruno Raffin[1,2,3] and François Faure[3,4,1]

[1]INRIA, [2]LIG, [3]University of Grenoble, [4]LJK-CNRS,

Abstract

Neighbor identification is the most computationally intensive step in particle based simulations. To contain its cost, a common approach consists in using a regular grid to sort particles according to the cell they belong to. Then, neighbor search only needs to test the particles contained in a constant number of cells. During the simulation, a usually small amount of particles are moving between consecutive steps. Taking into account this temporal coherency to save on the maintenance cost of the acceleration data structure is difficult as it usually triggers costly dynamics memory allocations or data moves. In this paper we propose to rely on a Packed Memory Array (PMA) to efficiently keep particles sorted according to their cell index. The PMA maintains gaps in the particle array that enable to keep particle sorted with $O(\log^2(n))$ amortized data moves. We further improve the original PMA data structure to support efficient batch data moves. Experiments show that the PMA can outperform a compact sorted array for up to 50% element moves.

1. Introduction

Particle systems are commonly used for simulating fluids, crowds, molecular dynamics, granular materials as well as impacts on concrete structures. These methods rely on particles whose state is updated according to interactions with neighbor particles (we consider here only short range interactions). Neighbor search is a computationally intensive step. Acceleration data structures are classically used to reduce its cost. A common approach consists in partitioning the simulated space with a regular grid, identifying the particles each cell contains. Cell size is defined according to the interaction range to limit neighbor search to the particles contained in the local and neighbor cells.

Given the large number of particles a simulation may require, performance can be significantly impacted by the memory layout. Having particles belonging to the same cell close by in memory enables to preserve a low cache miss ratio. But maintaining this locality as particles move may require costly memory copies or dynamic allocations. The challenge is thus to find a data structure that offers a good trade-off between particle locality and maintenance cost. To reach this goal it is important to take into consideration the strong temporal locality that usually show particle simulations: only a small ratio of particles actually change of cell between consecutive iterations.

In this paper we propose to store particles in a Packed Memory Array (PMA). Particles are stored sorted (according to their cell index) and evenly distributed in an array larger than the actual number of particles. To further improve spatial coherency, cells are indexed following a Z-curve pattern. An amortized scheme enables to maintain the PMA in a coherent state at a low cost. Inserting a new particle while maintaining the PMA state has a $O(\log^2(n))$ amortized cost. Range query or PMA scanning leads to a controlled cache miss overhead proportional to the PMA density. To optimize performance we adapt the PMA to move particles by batches, further reducing the cost of the PMA maintenance. Experiments show that the PMA can outperform a compact sorted array up to 50% elements moving amongst 10M ones. Performance is improved by a factor of 9 for 1% moves and by 50% for 50% moves. Experiments with a Smoothed Particle Hydrodynamics (SPH) simulation confirm the interest of the PMA data structure for keeping moving particles sorted.

The paper is organized in 6 sections. After reviewing related work (Sec. 2), we remind some basics about particle simulation and give an overview the PMA integration (Sec. 3). The PMA data structure is detailed (Sec. 4) with experimental results on random integers. An SPH simulation is then used to probe the PMA for particle simulation (Sec. 5). The conclusion summarizes our contribution and discusses future research directions (Sec. 6).

2. Motivation and Related Work

The naive approach for computing particle neighbors tests each particle pair, leading to $O(N^2)$ tests. The Verlet list [Fom11] improves this approach by using a neighborhood distance larger than the actual cut-off distance required by the simulation. Thus, the Verlet list, storing the neighbor list of each particle, does not need to be updated at every time step, amortizing its maintenance cost. Using a uniform grid to partition the simulation space, particles are distributed in each cell. Filling the grid is $O(N)$. Finding a particle's neighbors is $O(1)$ assuming that each cell can only contain a constant maximum number k of particles [Fom11]. To avoid costly memory allocations, usually an array of size k is allocated for each cell, making it a $O(k*m)$ size data structure, m being the number of cells. All cells, even empty ones, are stored, impairing memory consumption for very large simulation domains [IABT11].

Spatial Hashing proposes to improve memory consumption [THM*03]. Particles are distributed in a hash map according to their cell index. Filling the hash map is thus $O(N)$ and finding a particle's neighbors is $O(1)$. The authors note that having $2*N$ entries in the hash function enables to have a low collision rate. To avoid numerous dynamics memory allocations, the non empty cells are stored in a compact array with room for storing k particles per cell [IABT11].

Index sorting consists in sorting the particles in an array according to their cell index [MCG03]. The construction cost is thus $O(N)$. Memory usage is limited to the actual number of particles. If all cells store a pointer, set to null for empty cells or pointing to the cell first particle, finding a particle's neighbors is $O(1)$, but simulating large domains can be memory consuming. Memory can be traded for computations by storing only non empty cells, and performing a $O(\log(N))$ binary search to find the neighbor cells of a given particle.

Given the growing memory complexity of modern processors, being CPUs or GPUs, data layout in memory can significantly affect the cache hit or coalesced access ratios [TDR10]. Sorting particles to keep spatial locality, i.e. keeping the spatial locality when mapping the data in memory, can significantly decrease the required number of memory transfers. The Z-curve, also called Morton order, or Hilbert order are two classical memory layout patterns that show good locality properties. The Hilbert order often outperforms the Z-curve in term of locality preservation, but at the price of a more complex index computation. On the contrary, the Z-index can be efficiently computed by bitwise operations on the 3D coordinates [But71]. For particles simulations, the Z-curve is often used to improve the probability that data related to neighbor cells are stored close by and then are likely to be accessed consecutively. Thus, index sorting classically rely on a Z-curve. Ihmsen et al. [IABT11] improve spatial hashing, called compact hashing, by storing the actual particle data in a Z-order sorted array. In both cases, sorting is usually not performed at each iteration to amortize the sorting cost. Experiments in [IABT11] show performance improves significantly when relying on a Z-curve.

Particle simulations usually show a high temporal locality, i.e. only a small ratio of particles move between cells in consecutive time steps. Ihmsen et al. [IABT11] mention a 2% move ratio, while we experienced up to 25% moves with the fluid simulator [Hoe08]. One idea is to take advantage of this locality to build specific data structures that support an efficient incremental update, while still ensuring a fast neighborhood search and a low cache miss ratio.

The basic approach often reported is to rely on classical data structures and simply sort the particles periodically. For instance in [IABT11], particles are sorted once every 100 iterations, leading to a good trade-off between sorting cost and cache efficiency drop as the simulation shows only a 2% move ratio. As our experiments will show (Fig 6), sorting at every time step can be beneficial for higher move ratios.

[AE97] presents a general approach to make range query data structures dynamics. The idea is to handle several structures of varying 2^i sizes. When a new element needs to be inserted, all elements gathered in the smallest full j data structures are gathered in a $2^{(j+1)}$ size one. The cost of data structure building is thus amortized: the more elements the less often the data structure is rebuilt. This scheme is adapted to insertions but does not support efficient deletions. Following the same principle and inserting gaps amongst the elements of a sorted array, Bender introduced the library sort [BFCM06]. To support efficient element removals as well as insertions in a sorted array, the data structure was further modified leading to Packed Memory Arrays (PMA) [BDFC05,BH07]. A controlled gap density is maintained into an array of sorted el-

ements enabling to insert or delete an element in $O(\log^2(N))$ amortized element moves. These gaps obviously require extra storage space and generate additional memory transfers during scans. The PMA thus enables to control the trade-off between maintenance cost and memory dispersion. PMAs have been developed for large databases in conjonction with Cache Oblivious B-Trees [BDFC05], and, to our knowledge, never used for numerical simulations. We propose in this paper to keep particles Z-sorted using a PMA. In simulation, contrary with databases, the updates are applied in large batches at the end of each time step. Thus we extended the PMA to support batch insertions, deletions and moves.

3. Overview

Before detailing the PMA and its application to particle sorting, we review some of the basics of particle simulations and give an overview of the PMA use with particles. Particle simulations classically perform first a neighbor search, based on a cut-off distance. The entire list of particles is next scanned several times to update particle states. For most particle-based simulations, (Alg. 1), densities and next forces are updated based on particle neighborhoods, before the integration and position update that only require to access particle local data. A naive neighborhood search is thus $O(N^2)$ while all other scan steps are $O(N)$.

Algorithm 1 Particle simulation update

```
State neighborhood search
for all particles do
  compute density, pressure from neighborhood
for all particles do
  compute force from interaction with neighbors
for all particles do
  update positions
```

As mentioned in previous section, a classical approach to accelerate neighborhood search is to divide the simulation space into a uniform 3D grid of *cells*. Each particle is assigned to the cell its centroid belongs to. Finding the neighbor particles only requires to test the particles belonging to the current and neighbor cells.

Particles belonging to the same cell can simply be linked. Though it avoids data movements, it can seriously affect cache performance. The particles of a cell are accessed together. If the particles of a cell are spread in memory, accessing them may potentially lead to load numerous cache lines with a low relevant content (one particle per cache line in the worst case). To avoid this issue, particles can be sorted according to their cell index (Fig 1). To further improve memory locality, cells can be indexed following a Z-curve. Thus the particles of spatially close cells will tend to be stored nearby in memory. Though cache efficiency is increased, sorting particles introduces an extra cost. With an array of contiguous particles, moving one of them can lead to shift every other element, whatever the method we choose (re-sorting the whole array with standard sorting algorithms, or moving particles using an insertion sort like pattern).

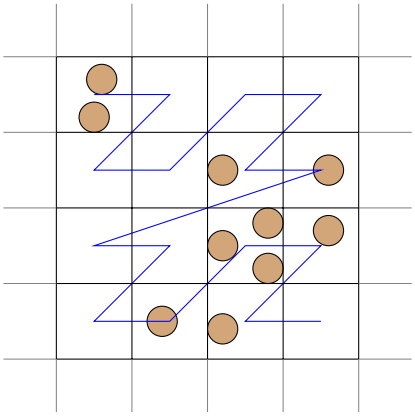

Figure 1: *Particles (in brown) placed into a collision detection grid in 2D. Cells are Z-curve ordered (blue segmented line).*

Inside an array, the only way to reduce the amount of memory moves is to anticipate insertions keeping extra (empty) spaces. With enough gaps, there is always an empty space close to the target insertion location that can be used to make room for the element to insert with a low number of memory moves. The space density should nevertheless be limited to avoid increasing the memory consumption and memory transfers. Indeed, these gaps will obviously make the overall array bigger and decrease the ratio of the relevant data a loaded cache line can contain.

The PMA structure implements this apparently simple idea with an amortized scheme for keeping a bounded gap density as elements are removed or inserted. As already mentioned we propose to use the PMA to store particles and keep them sorted according to their cell index.

4. PMA

We present the PMA algorithm as introduced in [BDFC05, BH07]. A PMA consists in an array of size N. *Gaps* are empty array elements, opposed to filled array elements, called *elements*, that actually contain the sorted data. The PMA is evenly divided into

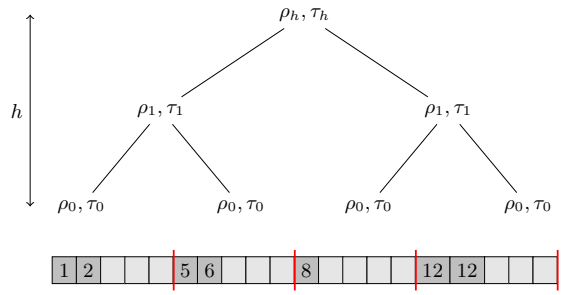

Figure 2: *PMA representation with $S = 4$ segments of size 5. Every level of the segments tree owns its minimum and maximum element densities.*

$\Theta(N/\log N)$ pieces called *segments* (Fig. 2). The density of segments, i.e. the ratio of elements against gaps, is maintained between the density bounds ρ_0 and τ_0. The density of a pair of consecutive segments, level $l = 1$, is maintained between ρ_1 and τ_1. This scheme repeats by doubling the number of segments considered up to include the full array bounded by densities ρ_h and τ_h. Densities are linearly staged between the different levels. At level l, the density of a *window* of 2^l segments is bounded by:

$$\tau_l = \tau_h + (\tau_0 - \tau_h)\frac{(h-l)}{h} \quad (1)$$

$$\rho_l = \rho_h + (\rho_0 - \rho_h)\frac{(h-l)}{h} \quad (2)$$

The density gap between two consecutive levels l and $l+1$ is $O(1/\log N)$ to ensure the complexity of the algorithm [BH07].

Extra constraints on maximum and minimum densities ensure that:

$$\rho_0 < ... < \rho_h < \tau_h < ... < \tau_0 \quad (3)$$

Thus, if a window of size l respects its density bounds, then all sub-windows also respect their density bounds.

When inserting a new element inside a PMA (Alg. 2), the destination segment is first found. Before proceeding to the insertion, we check that the target segment density, counting the element to insert, respects the segment upper density bound τ_0. If not, we look for the smallest windows at level l enclosing the segment that respects its upper density bound τ_l. We next proceed to an operation called *rebalance* that shares out the gaps among the different segments in that window and insert the new element. A rebalance consists in two steps. First elements are packed at one end of the window. Then they are moved into their destination segment, by one scan of the concerned segments. The obtained PMA respects all density bounds.

If no window can be found, i.e. τ_h is not respected, the PMA size is doubled and rebalanced. Element deletion is similar except that the ρ_l densities must be considered. If the whole PMA becomes too empty, i.e. ρ_h is not respected, the PMA size is divided by two. To make sure that when halving or doubling the array size, the new PMA is within the density bounds, the extra constraint on densities must be respected:

$$2\rho_h < \tau_h \quad (4)$$

Algorithm 2 Pseudo algorithm for one element insertion inside a PMA.

```
binary search the segment s the new element should go into
if τ0 exceeded then
    search for the l window that fits τl
    if τh not respected then
        double PMA, l ← h
        rebalance elements
    else
        rebalance window of 2^l segments
    end if
end if
search and insert element in the 2^l segments window
```

This scheme ensures that any insertion or deletion is performed in $O(\log^2 N)$ amortized element moves and that for special patterns, like random insertions, the cost is reduced to $O(\log N)$ [BDFC05, BH07].

The PMA is a cache-oblivious data structure: the PMA is built without taking into account a given cache line or total cache size. The performance is anyway guaranteed, with $O(1 + S/B)$ memory transfers when scanning S consecutive elements on a machine with a B size cache line. Cache-oblivious data structures have the advantage, over cache-aware ones, to be more flexible. For instance, they usually can be efficiently shared between a GPU and a CPU.

4.1. Batch Moves

We now present how we extended the PMA to improve the performance of moving batches of elements. An element is moving when its key, i.e. cell index, is changed. A move can be implemented by first removing and next inserting this element. However this may lead to rebalance twice the same window inside the PMA. When several elements are moving at the same time, the chances that some array section get rebalanced several times increases. To save memory movements we propose to apply moves batchwise.

First, moving elements are removed from the PMA without rebalance. A gap is simply set where the element to remove is. Moving elements are copied in

a temporary array, called *moving array*. All moving elements are then processed together.

The moving array is sorted using a classic sort algorithm like qsort. We then consider the first element of the PMA middle segment. The position of this pivot is searched (binary search) in the moving array. The moving elements on the left of the pivot need to be inserted in the first half of the PMA, the others in the right part. We then test the density against its allowed bounds, counting the elements to insert, for each of these 2 windows delimited by the pivot.

If the window density does not respect the bounds, its rebalance is triggered. The elements to insert as well as the element in the PMA are already sorted. The rebalance can then be implemented in a single scan on the window, using an extra temporary array to store elements that need to be shifted forward to leave room either for an element to insert or to a gap.

If the density bounds are respected, the process is recursively applied down to individual segments as long as no rebalance is triggered in a larger window. As long as elements need to be inserted, we need to check the window density against its upper and lower bound. Since moving element were previously removed from the PMA without rebalance, some window densities may be below the lower threshold, even if some elements need to be inserted. Once no more element need to be inserted in a window, only the lower density bound needs to be probed.

This approach equally applies to raw suppressions or new insertions. New elements to insert are sorted with the moving elements in the moving array. The recursive process starts one level higher, testing the density of the full PMA to check if halving or doubling the PMA size is necessary.

In case of random uniform moves, rebalance will take place on the smallest windows, i.e. at segment level, with a high probability. The move cost is dominated by the sorting of the moving array and the rebalance of each segment. For a high move ratio, it is probably more efficient to simply sort all element in a dense array and next spread them in a PMA, thus saving the overhead associated with the recursive descent along the PMA windows.

4.2. Experimental Results

4.2.1. Density Bounds and Segment Size

The size of the PMA, N, is the smallest power of 2 greater than or equal to K/τ_h, where K is the number of valid elements to be sorted in PMA. The number of segments, S, is the smallest power of two larger than $N/\log(N)$ as suggested in [BFCM06]. Thus the size of a segment is given by N/S. The minima densities for the segments (ρ_0) and the whole array (ρ_h) are chosen by symmetry around 0.5 from the maxima ones (τ_0 and τ_h resp.). The densities ρ_h and τ_h must also comply with the constraint $2\rho_h < \tau_h$ (Eq. 4).

K	qsort	isort	PMA
100 000	13.4±0.06	1247±5	3.86±0.02
1 000 000	169±0.6	xx	38.9±0.06
10 000 000	2088±8	xx	372.2±0.5

Table 1: *Average update times (ms) and standard deviation for quicksort (qsort), insertion sort (isort) and PMA with 5% of moves applied to K sorted elements.*

No explicit parameter space exploration was performed but we tested various maxima densities τ_0 and τ_h for various K values. We observed in particular that given a number K of elements, τ_h should be chosen such as the final density is between 50% and 95%. When the final density is lower than 50%, there are too many gaps with respect to the number of elements, elements are then spread out accross a larger memory space, creating more memory transfers. Moreover, the size of the PMA being bigger (actually, the double) than the optimal one, it leads to more segments, thus extra PMA maintenance. On the other hand, if the final density is higher than 95%, the segments are almost all full. The insertion is more likely to cause rebalance at a high level.

In the experiences presented in next sections, we take $\tau_0 = 0.92$, $\tau_h = 0.7$, $\rho_h = 0.3$, $\rho_0 = 0.08$ as these values gave good results for different PMA sizes.

4.2.2. PMA Evaluation

We compare the performance of the PMA against a dense array sorted using a quicksort and an insertion sort for different percentages of moves. All data structures are initialized with random sorted integers (uniform distribution). A defined percentage of elements are randomly elected and assigned new (random) keys. The dense array is sorted again with a quicksort or insertion sort. A batch move is executed on the PMA. We measure the time taken to update all these data structures.

The CPU used is a 4-core Intel Xeon E5520 @2.27GHz (only one core is used for running a simulation) with 18Go of memory. The compiler used is g++-4.6.2. We average execution times over 32 passes.

We use the highly optimized implementation of the quicksort algorithm provided by the libc, *qsort*, and a home made insertion sort code, *isort*. Table 1 gives average execution times for 5% moves on various number of elements. The PMA outperforms the quicksort

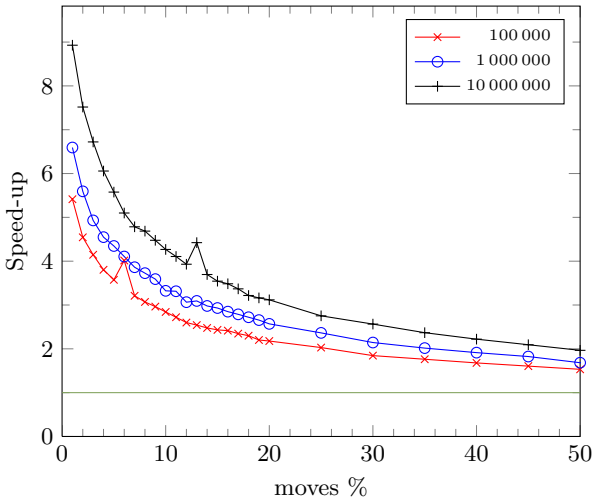

Figure 3: *Speed-up of PMA versus qsort against % of moves, for* 100 000*,* 1 000 000 *and* 10 000 000 *elements. The speed-up is the ratio of the sort execution time using qsort to the sort execution time using the PMA.*

and the insertion sort. The insertion sort, often recognized as the most efficient sort for almost sorted arrays, is not competitive even with the quick sort. The insertion is efficient for almost sorted arrays for K small. But for large K values as the one tested here, the cost of data movements becomes prohibitive. Thus experiments were not performed for insertion sort with $K \geq 1\,000\,000$.

Figure 3 presents the results for a move ratio ranging from 1% to 50% on 100 000, 1 000 000 and 10 000 000 integers. The results show that with 10 000 000 elements, the PMA is about 9 times faster than qsort for 1% moves. With 50% moves, the speed-up is 2. As expected, the PMA performance decreases along with the percentage of moves. Also notice that the PMA efficiency increases along with the number of elements.

4.2.3. PMA Overhead

If gaps enable to efficiently maintain moving elements sorted, they also introduce an overhead during range queries. Compared to a dense array, the valid elements are spread on a larger memory region, causing extra memory transfers, and a check is required at every location to probe the element validity (*validity check*). We evaluated this overhead through a scan operation on all the elements of a PMA in the following way: we build a PMA with a set of random elements and we benchmark the time to compute the sum of all the elements. We compare to the time needed for summing the same elements stored in sorted dense array.

K	array	PMA	ratio
100 000	0.21	0.78	3.7
1 000 000	2.2	8.3	3.7
10 000 000	22.3	68.8	3.1

Table 2: *Execution time (ms) and performance ratio (ratio) for summing K elements stored in a dense array (array) and a PMA.*

A direct implementation (one loop iterating on the PMA and a validity check for each element) leads to a poor performance. Beyond the expected overhead, we suspect that the validity check impairs some compiler optimizations and cause stalling in the processor pipe-line. After experimenting several approaches, the most efficient one consists in storing the element validity in a bitmask array. When reading one element of this bitmask array, we get 64 bits encoding the validity of 64 PMA elements. The scan is then written using 2 loops, an outer loop iterating on the bitmask elements, i.e. considering 64 PMA elements per iteration, and an inner loop probing the validity of these 64 elements. For $K = 1\,000\,000$ elements (Table 2), stored in a PMA array of size $N = 2\,097\,152 = 2.09K$, the scan is 3.7 times slower than the scan on a dense array. For $K = 10\,000\,000$ elements, the size of the PMA is $N = 16\,777\,216 = 1.7K$, explaining the improved performance ratio (3.1 for a fill ratio of 59% instead of 3.7 for 48%).

The sum operation performed per element is lightweight. More time consuming operations will make the overhead less noticeable. If many range queries operations need to be performed, like for a fluid simulation, other approaches can be considered to amortize this overhead. As we discuss in Section 5.2, we build an indirection array pointing to the PMA valid elements to save on the validity check.

5. Particle Simulation with the PMA

Next step is to use the PMA for a particle simulation. We tested the PMA in *Fluids* [Hoe08], an Open Source fluid simulator. We detail implementation and performance issues in the following.

The native *Fluids* implementation relies on a double array, one storing all cells, even empty ones, and a second one storing particles. By default particles are not sorted. *Fluids* fills the cells through a scan, each cell pointing to its first particle, while the remaining particles of the cell are chain linked, this operation is called *particles insertion in grid*.

We replaced the particle data array by a PMA that takes care of maintaining the particles sorted. To avoid

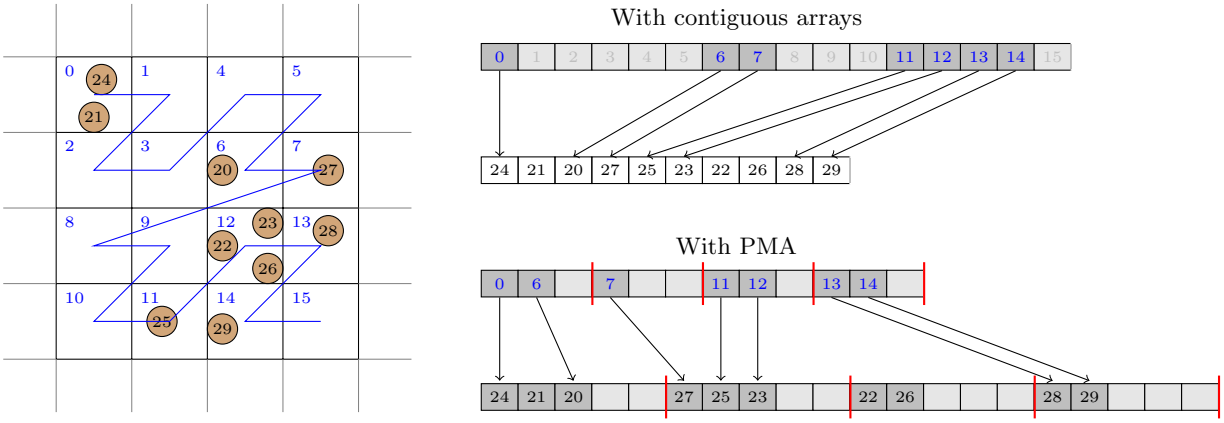

Figure 4: *Collision detection grid in 2D with particles (in brown) sorted according to their cell index (blue Z-curve) in two dense arrays (top right) or using two PMA data structures,one for storing each non empty cell and a pointer toward the cell first particle, and one for the particles (bottom right).*

size, #nb	tested pairs	valid pairs	ratio	CD
$2h, 8$	36878962	2691708	13.7	298
$h, 27$	16472058	2691708	6.1	232
$h, 14$	8236029	2691708	3.1	145

Table 3: *Impact of the cell size (size), and of the number of neighbor cells tested (#nb) on the number of tested pairs and the collision detection (CD) time (ms) with native Fluids implementation.*

the creation of a big array for all cells of the simulation space, we can also use a PMA to store non empty cells (Fig. 4). The batch insertions and deletes described in section 4.1 are used to keep up to date the non empty cells. The sizes of the arrays are thus only dependent on the particle number, and not on the simulation domain size.

We implemented a Z-curve cell indexing used to sort the cells and particles. A parameter enables to control the frequency of particle sorting for the native *Fluids* code, while the PMA maintains the particles sorted at each iteration.

As mentioned by [IABT11], sorting particle indices first and then updating accordingly the actual particle data can improve performance. This optimization could be implemented for both cases, but the experiments presented here rely on direct sort of particle data.

Fluids originally uses a cell size of 2 times the influence radius h. This choice allows to test only 8 neighbor cells per particle, provided that we take into account the position of the particle inside the cell.

We implemented a more classical grid, with a cell size of h that requires to test for proximity the particles of the 27 neighbor cells. We can further reduce the number of particle pairs to test by taking into account the symmetry of interactions. Indeed, we can test only 14 neighbor cells to find only once each pair. In that case, we need to update the neighbors of both interacting particles. Table 3 presents experimental results comparing the different methods for a scene of 130 000 particles, using the standard contiguous arrays CD from *Fluids*. The collision detection execution time can be divided by 2 when testing only the particles of 14 cells compared to the initial Fluid implementation testing the content of 8 larger cells.

5.1. Performance Results

Figure 5 presents the execution time of the sort operation during a corner breaking dam (CBD) simulation with 2 900 000 particles and 1 830 180 cells (354x94x55). The PMA performance is compared to the fluid simulation code in the situation where particles are sorted at every time step, called *reference implementation*. All cells (empty or not) are stored in a dense array.

The reference sort implementation is organized in four steps. A first pass builds an array of pairs consisting of every particle position in memory and their key according to the Z-curve. Qsort is then applied to that array and not on the whole data array. Then a temporary data array is used to copy the particles to their target position in memory. Finally, we copy the temporary array to the main particle data array. This saves time because the sort itself is, in this case, applied to smaller element sizes. In spite of this optimiza-

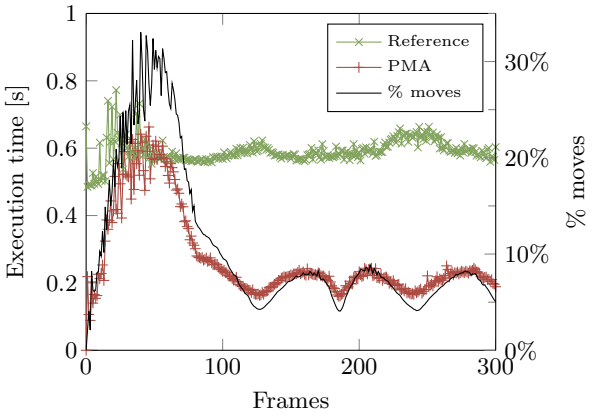

Figure 5: *Sort execution time (s) and move ratio (%) per iteration during a CBD simulation of* 2 900 000 *particles, for Fluids with particles sort at every time step (Reference) and for the PMA.*

step	%
sort	4.5
detect neighbors	49.1
compute pressures	6.3
compute forces	36.8
update positions	3.2

Table 4: *Relative weights of the different steps of the simulation. % values are averaged over* 4000 *time steps simulating* 180 000 *particles with Fluids code sorting particles at every time step.*

tion, which benefits only to the reference implementation, the PMA is three times faster than the reference implementation for about 10% of particle moves. Performances are similar around 20% of particle moves.

The corner breaking dam simulation starts with particles falling down till they reach the ground. At this time, particles move quickly, leading to a high move ratio. With the stabilization of the system (here after about 100 frames), the number of moves decreases. We globally experienced with the Fluid simulator a high level of particle moves compared to the 2% discussed in [IABT11]. Particle simulations with such a low level of particle move will significantly benefit from using a PMA data structure.

5.2. PMA Overhead Impact on the Whole Simulation

Table 4 presents the relative execution times of the different steps of a CBD simulation of 180 000 particles. Here the collision detection execution time is an order of magnitude higher than the sort execution time. This is again a noticeable difference with [IABT11], where the authors even discuss the possibility of recomputing neighbors several time per time step rather than storing them. Such approach is of course not relevant with such a high cost for collision detection.

This high collision detection cost versus sorting time is also not favorable to the PMA. The overheads absorb all the benefit of the fast particle sorting. The timing of the different steps reveals that PMA overhead is almost not noticeable for the pressure and force computations neither for position updates. The overhead is small compared to the time consuming operations executed for every particle. However, we initially experienced a high overhead during the collision detection step. To reduce this cost, we build an indirection array of PMA valid positions before the beginning of the neighborhood search. The goal is to remove the validity test that can prevent compiler optimizations and filling the processor pipe-line, and replace it with an access to an indirection array pointing to the valid elements in the PMA.

If particles are sorted in memory, it is easy to insert particles in the different cells in one single scan because particles of the same cell are contiguous in memory. Thus we can use the indirection array to store only the first and the last particle of each non empty cell. This latter improvement is suitable as well for *Fluids* as long as the particles are sorted in memory. It saves 12% of time for the whole collision detection (particles insertion in grid and neighbor search) for *Fluids*. Combined with the use of the indirection array, it saves 35% for the PMA.

Figure 6 presents the total execution time for simulation with 180 000 particles and 132963 cells (141x41x23). The PMA performance is compared to the native fluid simulation when we never sort particles in memory (*no sort*), when particles are sorted every 100 time steps and when they are sorted every time step. It is clear that sorting particles improves significantly the performance of the native *Fluids* code, showing the benefit of maintaining a good memory locality. On the other hand, for the whole simulation, sorting every time steps saves 5% of the computation time against sorting every 100 time steps. This is another difference we noticed compared to [IABT11] where they get the best trade-off between cache efficiency and sort overhead by sorting only every 100 time steps. Finally, using the PMA gives the best performance, saving 4 extra % of the total execution time.

6. Conclusion

In this paper, we proposed to store particles in a Packed Memory Array. A PMA maintain gaps be-

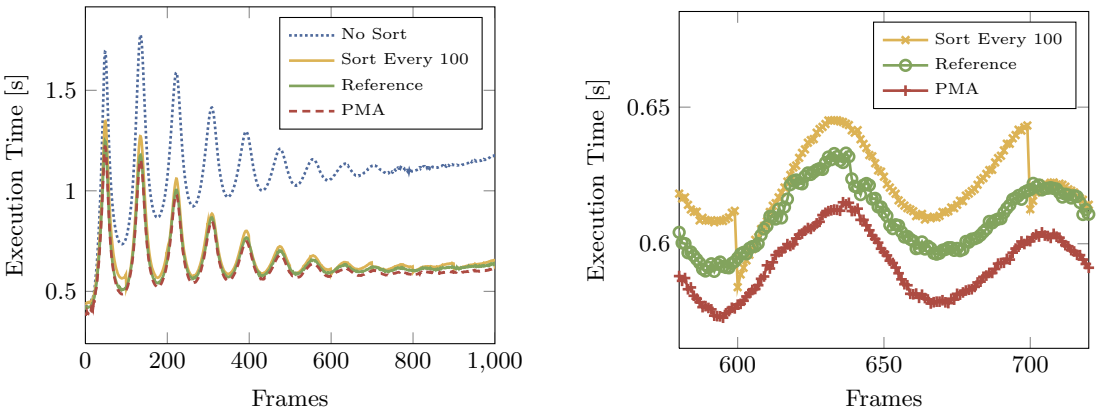

Figure 6: *Total execution times (s) for a CBD simulation of* 180 000 *particles with native code when particles are never sorted in memory after initialization (No Sort), when they are sorted every* 100 *time step (Sort Every* 100*), every time step (Reference), and using the PMA. On the left, we show the execution time for* 1 000 *time steps. On the right, we focus on the* $580 - 720$ *range to highlight differences between sort politics.*

tween particles, reducing significantly the cost of moving particles. The PMA significantly outperforms a classical quick sort for a reduced ratio of particle moves, a common situation for particle simulations that show a high level of temporal coherency. The PMA thus appears as a trade-off between a dense array storage leading to costly element moves, and linked arrays requiring either a large, often unused, amount of memory, or costly dynamics memory allocations. Experiments with a fluid simulator code show that the gain for sorting can be significant, even if the overall performance benefit is limited due to the reduced weight of sorting.

The mechanism could benefit as well to other particle-based simulations such as crowds, molecular dynamics, granular materials, or boids simulations. Future work also include evaluating the benefits of PMAs for other particle based simulations, and developing parallel PMA implementations for multi-core CPUs and GPUs.

Acknowledgments

We would like to warmly thank Swann Perarnau for his contribution to code development. This work was partly funded by ANR projet Repdyn 09-COSI-011-05.

References

[AE97] Agarwal P. K., Erickson J.: Geometric range searching and its relatives. In *Advances in Discrete and Computational Geometry* (1997), American Mathematical Society, pp. 1–56.

[BDFC05] Bender M. A., Demaine E. D., Farach-Colton M.: Cache-oblivious b-trees. *SIAM J. Comput. 35*, 2 (2005), 341–358.

[BFCM06] Bender M. A., Farach-Colton M., Mosteiro M. A.: Insertion sort is o(n log n). *Theor. Comp. Sys. 39*, 3 (June 2006), 391–397.

[BH07] Bender M. A., Hu H.: An adaptive packed-memory array. *ACM Trans. Database Syst. 32* (November 2007).

[But71] Butz A.: Alternative algorithm for hilbert's space-filling curve. *Computers, IEEE Transactions on C-20*, 4 (april 1971), 424 – 426.

[Fom11] Fomin E. S.: Consideration of data load time on modern processors for the verlet table and linked-cell algorithms. *Journal of Computational Chemistry 32*, 7 (2011), 1386–1399.

[Hoe08] Hoetzlein R. C.: Fluids v2.0, open source, fluid simulator, 2008. URL: `http://www.rchoetzlein.com/eng/graphics/fluids.htm`.

[IABT11] Ihmsen M., Akinci N., Becker M., Teschner M.: A parallel sph implementation on multi-core cpus. *Computer Graphics Forum 30*, 1 (2011), 99–112.

[MCG03] Müller M., Charypar D., Gross M.: Particle-based fluid simulation for interactive applications. In *Proceedings of the Symposium on Computer Animation* (2003), SCA '03, Eurographics Association, pp. 154–159.

[TDR10] Tchiboukdjian M., Danjean V., Raffin B.: Binary Mesh Partitioning for Cache- Efficient Visualization. *IEEE Transactions on Visualization and Computer Graphics 16*, 5 (2010), 815–828.

[THM*03] Teschner M., Heidelberger B., Mueller M., Pomeranets D., Gross M.: Optimized spatial hashing for collision detection of deformable objects. In *Vision, Modeling, Visualization (VM 2003)* (November 2003), pp. 47–54.