High Performance Algorithms for Counting Collisions and Pairwise Interactions

Matheus Henrique Junqueira Saldanha[0000−0001−7701−5583] and
Paulo Sérgio Lopes de Souza[0000−0002−1560−2704]

Institute of Mathematics and Computer Sciences, University of São Paulo
mhjsaldanha@gmail.com, pssouza@icmc.usp.br

Abstract. The problem of counting collisions or interactions is common in areas as computer graphics and scientific simulations. Since it is a major bottleneck in applications of these areas, a lot of research has been carried out on such subject, mainly focused on techniques that allow calculations to be performed within pruned sets of objects. This paper focuses on how interaction calculation (such as collisions) within these sets can be done more efficiently than existing approaches. Two algorithms are proposed: a sequential algorithm that has linear complexity at the cost of high memory usage; and a parallel algorithm, mathematically proved to be correct, that manages to use GPU resources more efficiently than existing approaches. The proposed and existing algorithms were implemented, and experiments show a speedup of 21.7 for the sequential algorithm (on small problem size), and 1.12 for the parallel proposal (large problem size). By improving interaction calculation, this work contributes to research areas that promote interconnection in the modern world, such as computer graphics and robotics.

Keywords: collision count · pairwise interaction · GPU · high performance computing · parallel computing · algorithm

1 Introduction

As the performance growth of a single processor core decreases, attention has been shifting towards other means to decrease execution time of programs. Given the lower price of primary and secondary memory, compared to processors, exchanging memory usage for a reduction on algorithm complexity is an interesting option. Another commonly explored alternative is to exploit the increasing parallelism available in hardware, which can be accomplished by parallelizing existing sequential algorithms. This does not always work well, and a good parallel algorithm might only be brought forth by a complete re-analysis of the problem and design of new parallel algorithms. In this article, solutions that explore these two options are provided to accelerate the problem of calculating interactions (e.g. forces, contacts) in N-body environments.

The problem of interaction counting is widely present in areas that promote interconnection in the modern world: computer graphics and virtual reality, that

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_14

connect people through games and link computer science to medicine, for example, where virtual environments can be used for practicing surgery; robotics, that interconnects objects and robots; and scientific simulations, that connect computing with areas that need to simulate complex natural phenomena such as protein folding and planet motion. Despite being largely used, interaction counting is a major bottleneck [13] in many applications. It may come in two flavors: 1) interactions might only matter within subsets of objects (e.g. collisions, where only neighbor objects are relevant), and it is common to use strategies such as spatial partitioning [6] or bounding volume hierarchies [19] to find these subsets before performing interaction counting; or 2) all interactions matter (e.g. gravitational forces), where it is common to use techniques that allow a set of objects to be regarded as a single object, such as the Fast Multipole Method [8]. Besides that, it is also often the case where the objects in question are in motion and collision detection must be done every small time steps, such as in a robot calculating collision-free paths. For such situations, besides the aforementioned strategies, time and space coherence is also used, that is, the fact that objects won't move too much in a short time span is exploited [9]. In any of these cases, however, there will still be a phase where smaller sets of objects undergo interaction calculation, and the usual way is to iterate over each object and compare it with the others, giving a $O(N^2)$ complexity.

Even though a lot of research has been carried out on alternative strategies to amortize the cost of this pairwise-comparisons $O(N^2)$ approach, not much has been done regarding this brute-force algorithm itself. This paper proposes a new parallel algorithm for calculating interactions that is designed to make better use of architectural resources on GPUs. The algorithm is mathematically proved to be correct, and results show a speedup of 1.12 over parallelization of the straightforward approach. We also propose a sequential algorithm for counting collisions among punctual objects, which has a $O(N)$ complexity at the cost of high virtual memory usage, and experiments show a speedup of 21.7 for limited size objects. By accelerating the pairwise-comparisons algorithm, we hope to facilitate smoother animations and virtual reality environments, simulations that take less time (or provide better results in the same amount of time), and robots that can better avoid collisions with its surroundings.

The document is organized as follows. An overview of research in collision counting is discussed in Section 2. In Sections 3 and 4, two algorithms for collision counting are proposed. Experiments with the proposed algorithms are discussed in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

Calculating interactions is a common problem that has been the subject of a vast amount of interesting research. Covering all of its facets is not our objective here, so in the following a short overview of the subject is provided.

I-COLLIDE [5] is a well-known suite of algorithms that perform fast collision detection among a large number of rigid or deforming objects, such as an en-

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_14

vironment with tens of thousands of triangles (that might compose an object). They focus on pruning potentially colliding sets (PCS) multiple times before performing the exact collision detection. CULLIDE [7] follows the same line and is parallelized for GPUs. Specializations of these algorithms for haptic systems, where collision detection must be performed thousands of times per second, were done in [10,11].

Spatial partitioning and bounding volumes hierarchies (BVHs) are broadly used for defining reduced sets of potentially colliding objects. A comparison among such techniques is found in [6], and k-d trees and octrees are notable in such problem. Reference [19] gives an efficient approach to construct BVH trees borrowing techniques from spatial partitioning, and [22] focuses on fast re-building of these trees on situations where there are moving objects.

Collision detection often involves calculating trajectories of objects, which is not always simple. In [21] the authors present fast and accurate methods for evaluating collision between triangulated models in such circumstances; their method involves a series of *coplanarity* and *inside* tests among elements (e.g. edges, vertices). In [18], where hair simulation is explored, there is a large number of hair-body collisions and hair-hair interactions (collisions and other forces such as friction and static attraction). In either case, the GPU approach we propose in this paper could be used to accelerate the calculation of interactions, such as the coplanarity tests. Similar studies are found in [3,4,16,17,20].

More related to engineering, in [23] is proposed a parallel algorithm for contact detection using spatial partitioning strategies, which is then experimented on simulation of concrete. Similarly, a framework that uses GPU for evaluating forces among particles of sand is given in [14]. Finally, in [15] the authors show an efficient parallelization, in the CUDA programming model, for the problem of evaluating gravitational forces among all bodies in an N-body system. They parallelized the straightforward sequential algorithm, where each body is compared with every other body. For summing these forces, in this paper is proposed a different sequential algorithm whose parallelization manages to make better use of GPU resources. To the best of our knowledge, this is the first work that brings an alternative to the pairwise-comparisons approach (for summing interactions) which provides benefits when parallelized.

3 A Sequential *O(N)* Approach

Consider the problem of counting the total number of collisions or contacts among beads (i.e. punctual objects) in a space $\mathcal{S}^3$ such that

$$\begin{cases} \mathcal{S}^3 = \mathcal{S} \times \mathcal{S} \times \mathcal{S} \\ \mathcal{S} = \{-a,\ -a+1,\ \ldots,\ a-1,\ a\} \subset \mathbb{Z} \end{cases}$$

that is, $\mathcal{S}^3$ is a three-dimensional, discrete and finite space whose axes are symmetric around the 0. Problems that do not suit such description are those that involve objects that have length, area or volume, possess non-discrete coordinates, or the range of possible coordinates is not bound by some known value

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_14

a. Although this excludes a large number of problems, there still remains some that fit in the given conditions. For example, in [1] proteins are represented as a chain of beads in $\mathbb{Z}^3$, so each bead is located at a limited distance from the previous one, which limits the coordinates to some interval, therefore allowing us to define a space $\mathcal{S}^3$ for the problem.

If the problem at hand can be made to fit into the aforementioned conditions, then the brute-force counting algorithm can be replaced by one with lower time complexity. First, the space $\mathcal{S}^3$ must be computationally represented in a way that each coordinate in $\mathcal{S}^3$ maps uniquely to a number stored in memory. This can be accomplished with a three-dimensional array of some numeric data type, which can be indexed using the coordinates of $\mathcal{S}^3$ themselves; however, to avoid indexing with negative numbers the space $\mathcal{S}^3$ must be translated to have its origin in (a, a, a). For simplicity, in what follows we consider that three-dimensional arrays can be indexed with negative numbers.

Algorithm 1: Counting number of collisions among a vector of beads.

```
int countColls(point3D beads[],
               int space[][][]){
  int collisions = 0;
  for (b in beads){
    int beadCnt = space[b.x][b.y][b.z];
    collisions += beadCnt;
    space[b.x][b.y][b.z] += 1;
  }
  return collisions;
}
```

Algorithm 2: Counting number of contacts among a vector of beads.

```
int countContacts(point3D beads[],
                  int space[][][]){
  int contacts = 0;
  for (b in beads)
   space[b.x][b.y][b.z] += 1;
  for (b in beads){
    contacts += space[b.x+1][b.y][b.z];
    contacts += space[b.x-1][b.y][b.z];
    contacts += space[b.x][b.y+1][b.z];
    contacts += space[b.x][b.y-1][b.z];
    contacts += space[b.x][b.y][b.z+1];
    contacts += space[b.x][b.y][b.z-1];
  }
  return contacts / 2;
}
```

Having defined the array that represents $\mathcal{S}^3$ in memory, we may then perform the counting algorithm. Algorithm 1 shows the main procedure for counting, with linear time complexity, the number of collisions among a vector of beads, each of which has three integer coordinates. In this algorithm, the number associated with a point of $\mathcal{S}^3$ represents the number of beads in that spot so far (assume, for now, that it is initialized with zeros). We begin by initializing the number of collisions with zero, and then iterate over the vector of beads. For each bead, we access the $\mathcal{S}^3$ space array using the bead's coordinates as the index, retrieving from memory the number *beadCnt* of beads in that place. We are on the process of adding a bead to a place that already contains *beadCnt* beads, thus generating *beadCnt* extra collisions that are added into *collisions*. Finally, we increment the number associated with that place in the space, to effectively add one new bead there. For initializing the space array, we would also iterate over the vector of beads, initializing only elements at the coordinate of each bead.

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_14

With another similar algorithm (see Algorithm 2), we can also calculate the total number of contacts among beads, that is, count how many pairs of beads are neighbors. As in the previous algorithm, the *space* array holds how many beads are placed in each coordinate of $\mathcal{S}^3$ so far, and we begin by initializing a *contacts* variable with zero. The first loop "places" beads in the *space* array, such that its element (x, y, z) has the number of beads with coordinates (x, y, z). Afterwards, we iterate over the vector of beads again, and for each bead b we fetch from *space* the number of beads in each of the six spots that are neighbors of b, then add them into *contacts*. There is still a problem to deal with: if beads b_1 and b_2 are neighbors, their contact is counted twice (in iterations for b_1 and b_2); because of this, the function returns *contacts/2*.

Determining the initialization pattern for the problem of contact counting follows a reasoning similar to the one used with collisions. The elements of *space* that are accessed are all neighbors of each bead, including the bead's own position, so they must be initialized to zero.

The presented algorithms involve 2 steps: initializing *space* and counting either contacts or collisions. Both steps consist of N iterations, one for each bead, and in each iteration we perform a fixed number of $O(1)$ instructions: sum, subtraction and memory load/store. Hence, the algorithms have $O(N)$ time complexity, so they tend to perform better than the quadratic approach for large enough N. However, experiments show that these algorithms are faster even for small N (< 64), which are elaborated in Section 5.

On the other hand, the algorithms make use of the three dimensional array *space*, whose size depends on the cardinality of $\mathcal{S}^3$. As defined earlier, $\mathcal{S} = \{-a,\ -a+1,\ \ldots,\ a-1,\ a\}$, so each axis of $\mathcal{S}^3$ has $2a+1$ elements, resulting in a total of $(2a+1)^3$ elements and consequently a $O(a^3)$ complexity of memory consumption. The a might be known on compilation time, in problems where the beads are confined in a box of known edge length. Another possibility is that a is a function of N. For example, in the case of proteins modelled as a chain of beads that begins in $(0, 0, 0)$, a chain of size N would require each axis to span from $-N$ to N, meaning a would be a function $a(N) = N$, and the memory usage complexity can be rewritten as $O(N^3)$.

Although there are cases in which the memory complexity is $O(N^3)$, this concerns only virtual memory. When counting collisions, for example, virtual pages are mapped physically only if they contain beads. In the worst case, each bead will be in a different page and N pages will be mapped, making it $O(N)$ in physical memory usage. An important consequence of this is that swap memory tends to be depleted before physical memory, so by the time swap depletes the counting program still will not have begun to access disk for swapping, which would greatly impact performance.

4 An Efficient Parallel Algorithm

The $O(N)$ approach tends to incur high memory consumption, which poses an obstacle to handle large problem sizes, in which case it is reasonable to use

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_14

parallel computing to distribute memory usage or computation among processing nodes. Applying this technique is possible for both the $O(N)$ and the $O(N^2)$ approaches, but due to the seemingly higher difficulty in using it for the $O(N)$ one, we analyze and propose an efficient parallelization of the $O(N^2)$ algorithm for counting any kind of symmetric pairwise interactions (SPI) among objects, not limited to collisions. The proposed algorithm is aimed mainly at GPUs, since it makes better use of its architectural characteristics, and experiments were performed using them. However, results are not limited to GPUs as there might be parallel architectures with similar characteristics and could be better exploited with the algorithm presented in the following.

In Algorithm 3, we present the standard sequential code for calculating SPI. For each object, we accumulate its interaction with all subsequent objects. Analyzing the nested loops in search for parallelization, we notice that they are not completely data-parallel due to the *interactions* variable, which is read and written in every iteration. Such variable is a reduction variable, which implies that parallelizing the iterations is still viable, provided that there is a reduction phase that agglomerates the intermediate results calculated by each parallel execution unit (denoted as *threads* from here on).

A second aspect of the standard sequential algorithm is that the outer loop is not balanced in terms of work executed per iteration. The first outer iteration executes $N - 1$ inner iterations, whereas the last outer iteration executes none. When we try to parallelize the problem, this could cause threads to be assigned different amounts of work, resulting in idle threads waiting for others to finish their larger burden. One way to promote balancing is to assign outer iterations to GPU threads in a round-robin fashion such that each thread performs at least two outer iterations. However, by agglomerating outer iterations this approach reduces the number of threads we can launch, and it also arguably reduces memory locality, which are undesirable properties for GPU algorithms.

Algorithm 3: Standard algorithm for calculating SPI.

```
for (i = 0 to N-1)
  for (j = i+1 to N-1)
    interactions += interact(obj[i],
                             obj[j]);
```

Algorithm 4: Proposed algorithm for calculating SPI.

```
for (i = 0 to N-1)
  for (j = 1 to (N-1)/2)
    interactions += interact(obj[i],
                             obj[(i+j)%N]);
```

For balancing the loop iterations in a way optimized for GPU, we offer an alternative solution whose main portion of code is shown in Algorithm 4. We now explain the idea behind it, prove that it works when N is odd and finally elaborate on how to make it work with even N too.

In the proposed algorithm, the outer loop can be seen as follows. Each bead i (we will use the term *bead* for simplicity, but it can be any kind of object) evaluates the interaction of itself with subsequent beads in a circular fashion. This circularity can be mathematically modelled by working in the universe of integers *modulo* N [12], which has interesting properties that we will use later:

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_14

$$a \equiv_N b \implies \forall k \in \mathbb{Z} \quad a + k \equiv_N b + k \tag{1}$$
$$a \equiv_N 0 \iff \exists c \in \mathbb{Z} \quad a = c.N \tag{2}$$

For each bead i we can now define two functions: $reach(s)$, that returns the index of the bead being evaluated in step s; and $reached(s)$ that returns the index of the bead evaluating bead i in step s. The algorithm begins in step $s = 1$ and goes forward until some stopping condition that we discuss now. For any bead i, we have:

$$\begin{array}{lll} s = 1 & reach(1) \equiv_N i + 1 & reached(1) \equiv_N i - 1 \\ s = 2 & reach(2) \equiv_N i + 2 & reached(2) \equiv_N i - 2 \\ & \ldots & \\ s & reach(s) \equiv_N i + s & reached(s) \equiv_N i - s \end{array}$$

where bead i evaluates beads j with j increasing, and bead i is evaluated by bead k with k decreasing as s advances (see Figure 1).

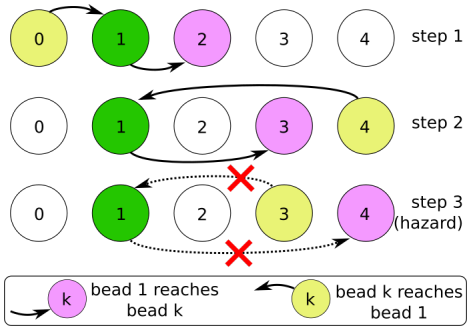

Fig. 1. Algorithm execution from the perspective of bead 1. Bead 1 reaches beads 2 and 3, and it is reached by beads 0 and 4 as s increases.

An undesired situation here is that bead i evaluates bead j when bead j has already evaluated bead i, so the same interaction would be evaluated twice. This is illustrated in Figure 1, where this situation happens in step 3, where bead 3 reaches bead 1, but bead 1 had already reached bead 3 in step 2. For an odd number N of beads, this hazard happens on the step where bead i reaches bead $j+1$ on the same step that bead j reaches bead i. Mathematically, the violation happens when $reach(s) - 1 \equiv_N reached(s)$, which for an arbitrary bead i gives

$$\begin{array}{lll} i + s - 1 \equiv_N i - s & & \\ 2s - 1 \equiv_N 0 & & \textbf{using (1)} \\ 2s - 1 = c.N & c \in \mathbb{Z} & \textbf{using (2)} \\ s = \dfrac{c.N + 1}{2} & \therefore \; s = \dfrac{N + 1}{2} & \end{array} \tag{3}$$

Note that modular arithmetic gives a set of answers instead of a unique one. It shows that the step s where the violation occurs is $(c.N+1)/2$ for some integer

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_14

c. If c is 0, then $s = 1/2$ which is invalid because, as stated earlier, the first step is $s = 1$; any $c < 0$ gives $s < 0$ which also does not make sense; $c = 1$ gives $(N + 1)/2$, which is in fact the first step where the violation occurs (N is odd, so the division results in an integer). Taking $c = 2, 3$, and so on yields higher steps where the violation would occur, but they do not matter for us because we will stop the algorithm before the first violation. Finally, since the first violation occurs at $s = (N + 1)/2$, we need to stop at the previous iteration, that is, at $s = (N - 1)/2$.

We still need to prove the equivalence between the proposed and straightforward algorithms. First, the number of different interactions among beads amount to $N.(N - 1)/2$, which is precisely the amount of interactions evaluated by the straightforward approach. We now prove that the proposed algorithm performs this same amount of work, and that all interactions evaluated are mutually different, hence proving the equivalence of both approaches.

Proposition 1. *The proposed algorithm evaluates $N.(N - 1)/2$ interactions among beads, for odd N.*

Proof. As stated before, each of the N beads evaluates its interaction with $(N - 1)/2$ subsequent beads, which amounts to a total of $N.(N-1)/2$ evaluations. □

Proposition 2. *All $N.(N - 1)/2$ interactions evaluated by the proposed algorithm are different from each other, for odd N.*

Proof. Take two arbitrarily different beads i and j, with $i < j$. Each bead evaluates the interaction between itself and subsequent beads, so for beads i and j one side of the interactions they evaluate is inherently different since $i \neq j$. Consequently, both beads would only evaluate the same interaction if it was the interaction among beads i and j themselves. Let us see when this happens.

For bead i to evaluate the interaction of i with j, it would be necessary that

$$\begin{aligned} reach(s) &\equiv_N j && \text{from bead i's perspective} \\ i + s &\equiv_N j \\ i + s - j &\equiv_N 0 \\ i + s - j &= c.N && c \in \mathbb{Z} \\ s &= c.N + (j - i) \ \therefore \ s = j - i \end{aligned} \tag{4}$$

For $c < 0$, s would be a negative value because $j-i$ (the distance between beads i and j) cannot be higher than $N-1$, and this is a contradiction because negative s never happens. If $c > 0$, then $s = c.N + (j - i) > N$, but s only gets values $1, ..., (N - 1)/2$. Finally, taking $c = 0$ makes $s = j - i$, which happens as long as $j - i$ also resides in interval $1, ..., (N - 1)/2$, that is, the distance between j and i is lower than or equal to $(N - 1)/2$. On the other side, bead j will evaluate its interaction with i when its $reach(s) \equiv_N i$, and developing this expression in a similar way as before we obtain $s = c.N-(j-i)$, but the only valid value for c is 1. This gives $s = N-(j-i)$, and $s \in \{1, ..., (N-1)/2\}$ only if $j-i \geq (N+1)/2$.

Therefore, let the distance between i and j be called $d = j - i$, then i will evaluate j only if $d \leq (N - 1)/2$, and j will evaluate i only if $d \geq (N + 1)/2$.

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_14

This means that i and j do not mutually evaluate each other, so the interaction among beads i and j is evaluated only once. This proves that all interaction evaluations are mutually different, completing the proof of equivalence between this approach and the straightforward one, for odd N. □

When N is even, a slight modification is needed[1]. The stopping condition of the outer loop is derived in a similar way as done before, but in this case the problem is not that the *reach*() and *reached*() arrows cross themselves; instead, they reach the same value. That is, for a given bead i we have $reach(s) \equiv_N reached(s)$, which will result, following the same mathematical steps as before, in $s = N/2$. This is the step in which bead i is evaluating some bead j while bead j is also evaluating bead i. To prevent this situation, we allow execution of $N/2 - 1$ steps normally, and the first half of the beads are made to execute one more. This works because in step $N/2$ the beads being reciprocally evaluated are on different halves of the vector of beads, since they are within a distance of $N/2$ from each other. The consequence of this modification is that the algorithm is not completely balanced any longer; some outer iterations execute one extra inner iteration.

This concludes the formulation of the alternative, balanced algorithm. This proposed approach has some nice theoretical properties, based on the concepts of *depth* and *work* [2]. Depth is the largest amount of work done sequentially by a single thread, while work is the total amount of work done by all threads launched. In the straightforward algorithm, we have a work-complexity of $N.(N-1)/2$ because that is the number of interactions that need to be evaluated, and a depth-complexity of $N-1$ because the thread that performs the first outer loop evaluates that many interactions. Note that we are ignoring the work and depth of the reduction phase because it is performed in the exact same way in both the straightforward and the proposed approaches. In our proposed algorithm, the work-complexity is maintained (seen in Proposition 1), while the depth-complexity becomes $(N-1)/2$ for odd N, and $N/2$ for even N. Therefore, we reduced the depth while preserving the amount of work, which indicates that if we had infinite physical processing units, the proposed approach could be faster.

In practice, both approaches can be parallelized by assigning each outer iteration to one thread, followed by a reduction phase where the threads cooperate to accumulate the intermediate results obtained. As was already mentioned, it is possible to parallelize the straightforward approach in a balanced way by distributing the outer loops over a smaller number of threads in a round-robin fashion. However, although this might perform well in distributed or multicore systems, it increases the algorithm's depth, reduces the number of threads that can be launched and degrades memory locality, which are not good properties for GPUs. Besides that, it also makes it considerably more difficult to manage usage of shared memory, which is a fast memory shared only by a block of threads. For these reasons, we have implemented in the CUDA programming model only the straightforward parallelization and the proposed one, and we show in Section 5 that the proposed approach is slightly better.

[1] Full formulation is available in mjsaldanha.com/articles/1-hpc-sspi/ .

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_14

5 Experiments and Results

In order to evaluate the performance of the $O(N)$ sequential counting algorithm of Section 3, we implemented both approaches, linear and quadratic, for counting collisions[2]. We designed the implementations so that they had similar characteristics to the protein structure prediction program (from [1]) analyzed and implemented in the broader context of this research. Hence, in each execution we perform the counting procedure upon multiple bead vectors, the space array is allocated only once, and each bead vector is generated by placing the first bead at $(0, 0, 0)$ and positioning the next bead in the neighborhood of the previous bead (similar to a protein), choosing any of the 6 directions randomly.

Figure 2 (left side) shows the experimental results. Each program was executed with varying problem sizes and each execution comprised counting the number of collisions for 1000 different bead vectors. For each problem size, we collected the wall clock time for each of 100 executions and took their mean. The total vertical length of the black error bars equals four standard deviations of the samples. These experiments were run using an Intel i7-4790 3.6GHz and 32GB of primary memory. For a problem size of 1920, which was the largest problem size that could be run in the system, the speedup was 21.7 and the linear approach required 52.82GB of virtual memory, as pointed in Figure 2 (left).

With the results shown in Figure 2 (left), the gain in execution time provided by the linear approach is clear. However, this comes at the cost of high consumption of virtual memory, which greatly limits the largest problem size that can be supported by one's system. In the case of the protein structure prediction (PSP) program we implemented[2] during the research project that revolves this paper, the number of beads among which collisions are calculated rarely exceeds 1000 (proteins rarely have that number of amino acids), and for this problem size the virtual memory usage is 7.5 GB, a feasible amount. Experiments with the PSP program showed speedups of 11.8 and 72.4 for proteins with 128 (a common size) and 768 amino acids, respectively, so by using the proposed algorithm the program was accelerated significantly. Possible reasons for the higher speedup are related to factors that are discussed below.

Some considerations must be made regarding the generation of beads in resemblance to the PSP algorithms. By allocating the space array only once in each execution, we reduce the cost of requesting memory from the operating system, which is present only in the linear approach; on the other hand, this means we do not reclaim virtual memory after using it for a single vector of beads, causing physical pages that were mapped to remain mapped until the end of the program execution. For bigger problem sizes, memory should be reallocated every K iterations so as to free unused allocated virtual pages and prevent the program to swap memory. In Figure 3 (left), we show what happens to the execution time when varying such number K of iterations. Notice that performance improves as we reallocate memory less often (due to lower memory

[2] In mjsaldanha.com/articles/1-hpc-sspi/ the reader can find the source code for all experimented programs mentioned in this article.

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_14

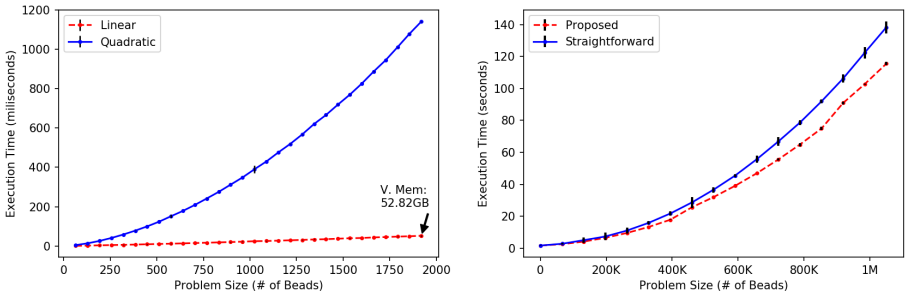

Fig. 2. To the left, execution time for linear and quadratic approaches for counting collisions. To the right, for counting collisions in GPU. All vertical black error bars have a length of four standard deviations of 100 samples taken.

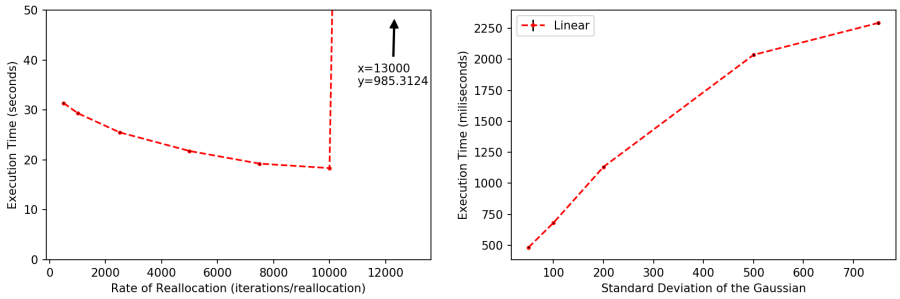

Fig. 3. Linear approach when: (to the left) varying the rate by which memory is reallocated, for a problem size of 13 000 vectors of beads; and (to the right) generating bead coordinates with a normal distribution with mean 0 and varying standard deviation, for a problem size of 1000 bead vectors.

management overhead), reaching an optimal point at a rate of one reallocation every 10 000 iterations. After that, the program begins to swap memory and performance degrades significantly.

A second consideration is that, by generating bead vectors as "random proteins", we are statistically confining beads to a smaller region around the origin of the space, hence improving cache usage and reducing the number of memory pages required. For better illustrating this, we show in Figure 3 (right) what happens to execution time if we generate the beads' coordinates using a normal distribution with mean 0 and varying standard deviation. As expected, performance degrades at higher deviation due to a lower rate of cache hits.

It follows from these two considerations that the frequency of reallocation of the space array, as well as the regularity of bead positions, must be taken into consideration when using the linear approach. There is a lot of room for analyzing such factors and how they apply to real applications; as that is not the objective of this paper, it is left as future work.

The parallel algorithms for SPI presented in Section 4 were implemented[2] in the CUDA programming model and evaluated using NVIDIA GPUs. Experiments used an NVIDIA Tesla P100 with 16GB of memory and 3584 CUDA cores

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_14

spread over 56 multiprocessors. Both the straightforward and the proposed approaches were optimized to achieve high occupancy and memory bandwidth, and all optimizations were applied to both approaches alike. Then, we executed each program 100 times for each problem size and took the mean of their wall clock execution times, which are shown in Figure 2 (right side); four standard deviations are represented by the black vertical error bars. Each program execution comprised calculating collisions for 100 randomly generated vectors of floating point spheres with a diameter of 1. Even though comparisons with sequential approaches could have been made, it is quite clear that parallel algorithms yield lower execution times and the objective here is mainly to show the benefits of the proposed parallelization compared with the straightforward one.

Figure 2 (right) shows that the proposed approach seems to perform well on GPU. In fact, for all problem sizes larger than 525 000, the proposed approach is more than 12% faster than the straightforward one ($p < 0.01$ using a t-test assuming unknown and different variances). The speedup is mainly due to two factors. First, in the straightforward approach each CUDA block is responsible for calculating interactions of a group of 1024 spheres with *all* subsequent spheres; because of this, some blocks perform more work than others, and in the last moments not all of the 56 multiprocessors are used, and the program is waiting for a few lengthy blocks to end their part of the work. Second, besides this block-level unbalance, threads within a warp (group of 32 threads that execute simultaneously) are also unbalanced, so when the warp is nearly concluding its work the 32^{nd} thread ends its job earlier and becomes idle waiting for the other threads to finish. In our proposed balanced approach, threads or multiprocessors that would otherwise be idle are put to perform useful work and contribute to finishing the interaction counting more quickly. Finally, if the problem in hand allowed the CUDA kernels to be launched in parallel, the GPU could hide a lot of the block-level unbalance; however, the warp-level one would remain negatively impacting the program speed.

6 Conclusion

Interaction counting is a usual problem, and when it needs to be performed, the quadratic pairwise-comparisons approach immediately comes to mind. For a long time this has been a significant bottleneck [13] on important areas such as computer graphics and scientific simulations; in the former, collision counting must be performed enough times per second to allow image frames to be delivered in a visually fluid way, and more frames mean more fluidity, so every millisecond matters; in the latter, simulations of galaxies or proteins may need a large number of iterations if a high level of reality is desired (possibly taking weeks to execute), so if the interaction counting performed every iteration is accelerated, either less simulation time would be required or more iterations could be performed in order to achieve better results. In either case, performing interaction counting in less time yields great benefits, which is why a lot of research has been done on the subject. However, research walked toward algorithms that focus on reducing the

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_14

number of objects among which interaction counting must be performed using the usual $O(N^2)$ approach, often in parallel.

In this paper two algorithms are proposed that aim at improving the pairwise-comparisons approach itself: a sequential approach with $O(N)$ complexity that works well for punctual objects in a limited discrete space, and a parallel approach that runs more efficiently on GPUs than the pairwise-comparisons algorithm's straightforward parallelization. These approaches can, of course, be used together with the algorithms that focus on pruning sets of interacting objects, in order to accelerate the phase where brute-force interaction counting must be performed. By using the $O(N)$ approach, interaction counting can be made significantly faster, at the cost of high memory consumption; our experience with accelerating a protein structure prediction algorithm, as already mentioned, shows a speedup of 72.4 using the same hardware. The proposed parallel algorithm may be used on large problems, with objects of any shape, for counting any kind of interactions, and experiments using GPU show that it can yield a 1.12 speedup.

A possible direction for future research is to evaluate possible benefits of parallelizing the proposed $O(N)$ algorithm in order to share the memory consumption among nodes, which would allow the algorithm to be used for bigger problem sizes. Also, the $O(N)$ algorithm is sensitive to cache effects and to how frequently memory is allocated and deallocated, so these factors should be further investigated, especially when applied to real problems. Besides that, since the proposed parallel algorithm has nice properties it could be analyzed on different parallel platforms so as to understand the limitations of the algorithm on each architecture; examples are Intel Xeon Phi processors and FPGAs.

7 Acknowledgement

We thank São Paulo Research Foundation (FAPESP) for funding this research project (grant 2017/25410-8, associated to 2013/07375-0), and the Center for Mathematical Sciences Applied to Industry (CeMEAI) for providing access to powerful computational resources.

References

1. Benítez, C.M.V., Lopes, H.S.: Parallel artificial bee colony algorithm approaches for protein structure prediction using the 3dhp-sc model. In: Intelligent Distributed Computing IV, pp. 255–264. Springer (2010)
2. Blelloch, G.E.: Programming parallel algorithms. Communications of the ACM **39**(3), 85–97 (1996)
3. Bridson, R., Fedkiw, R., Anderson, J.: Robust treatment of collisions, contact and friction for cloth animation. ACM Transactions on Graphics (ToG) **21**(3), 594–603 (2002)
4. Brochu, T., Edwards, E., Bridson, R.: Efficient geometrically exact continuous collision detection. ACM Transactions on Graphics (TOG) **31**(4), 96 (2012)

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_14

5. Cohen, J.D., Lin, M.C., Manocha, D., Ponamgi, M.: I-collide: An interactive and exact collision detection system for large-scale environments. In: Proceedings of the 1995 symposium on Interactive 3D graphics. pp. 189–ff. ACM (1995)
6. Elseberg, J., Magnenat, S., Siegwart, R., Nüchter, A.: Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. Journal of Software Engineering for Robotics **3**(1), 2–12 (2012)
7. Govindaraju, N.K., Redon, S., Lin, M.C., Manocha, D.: Cullide: Interactive collision detection between complex models in large environments using graphics hardware. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. pp. 25–32. Eurographics Association (2003)
8. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. Journal of computational physics **73**(2), 325–348 (1987)
9. Gregory, A., Lin, M.C., Gottschalk, S., Taylor, R.: A framework for fast and accurate collision detection for haptic interaction. In: ACM SIGGRAPH 2005 Courses. p. 34. ACM (2005)
10. Gregory, A., Lin, M.C., Gottschalk, S., Taylor, R.: A framework for fast and accurate collision detection for haptic interaction. In: ACM SIGGRAPH 2005 Courses. p. 34. ACM (2005)
11. Gregory, A., Mascarenhas, A., Ehmann, S., Lin, M., Manocha, D.: Six degree-of-freedom haptic display of polygonal models. In: Proceedings of the conference on Visualization'00. pp. 139–146. IEEE Computer Society Press (2000)
12. Knuth, D.E., Graham, R.L., Patashnik, O.: Concrete mathematics: a foundation for computer science. Adison Wesley (1989)
13. Lin, M., Gottschalk, S.: Collision detection between geometric models: A survey. In: Proc. of IMA conference on mathematics of surfaces. vol. 1, pp. 602–608 (1998)
14. Longmore, J.P., Marais, P., Kuttel, M.M.: Towards realistic and interactive sand simulation: A gpu-based framework. Powder Technology **235**, 983–1000 (2013)
15. Nguyen, H.: Fast n-body simulation with cuda. In: Gpu gems 3, chap. 31. Addison-Wesley Professional (2007)
16. Provot, X.: Collision and self-collision handling in cloth model dedicated to design garments. In: Computer Animation and Simulation'97, pp. 177–189. Springer (1997)
17. Redon, S., Kheddar, A., Coquillart, S.: Fast continuous collision detection between rigid bodies. In: Computer graphics forum. vol. 21, pp. 279–287. Wiley Online Library (2002)
18. Selle, A., Lentine, M., Fedkiw, R.: A mass spring model for hair simulation. ACM Transactions on Graphics (TOG) **27**(3), 64 (2008)
19. Stich, M., Friedrich, H., Dietrich, A.: Spatial splits in bounding volume hierarchies. In: Proceedings of the Conference on High Performance Graphics 2009. pp. 7–13. ACM (2009)
20. Tang, M., Kim, Y.J., Manocha, D.: Efficient local planning using connection collision query. Ewha Womans University, Korea, Tech. Rep (2010)
21. Tang, M., Tong, R., Wang, Z., Manocha, D.: Fast and exact continuous collision detection with bernstein sign classification. ACM Transactions on Graphics (TOG) **33**(6), 186 (2014)
22. Wald, I.: On fast construction of sah-based bounding volume hierarchies. In: Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on. pp. 33–40. IEEE (2007)
23. Zheng, J., An, X., Huang, M.: Gpu-based parallel algorithm for particle contact detection and its application in self-compacting concrete flow simulations. Computers & Structures **112**, 193–204 (2012)

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_14