Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Computational Physics 186 (2003) 697–703
JOURNAL OF COMPUTATIONAL PHYSICS
www.elsevier.com/locate/jcp

Short Note

A heap-based algorithm for the study of one-dimensional particle systems

Alain Noullez [a,*], Duccio Fanelli [b], Erik Aurell [b,c]

[a] CNRS, Observatoire de la Côte d'Azur, B.P. 4229, F-06304 Nice Cedex 4, France
[b] Department of Numerical Analysis and Computer Science, KTH, SE-100 44 Stockholm, Sweden
[c] Department of Mathematics, Stockholm University, SE-106 91 Stockholm, Sweden

Received 9 February 2001; received in revised form 20 December 2002; accepted 14 January 2003

Keywords: Event-driven simulations; Heap structure; One-dimensional gravitation

1. Introduction

In this paper, we discuss a fast algorithm which integrates numerically a one-dimensional system of N interacting particles, provided the dynamics can be Lagrangian integrated between two successive collisions. An important application is self-gravitating systems in one dimension, but similar models with Lagrangian invariant or quasi-invariant force fields can also be treated.

One-dimensional systems have the important characteristic that the set of positions is well-ordered. This means that all $N-1$ possible collisions between N particles can be easily enumerated and that the neighbors of two colliding particles can be found in $\mathcal{O}(1)$ operations if we keep the particles sorted by position. It is then possible to build an event-driven algorithm to simulate a set of particles by finding the minimum of all possible collision times, evolving all particles up to that time and repeating the procedure [4,16]. At first sight, and in all already published solutions, this involves $\mathcal{O}(N)$ operations per collision. However, in one dimension, it is possible to update only the states of the two colliding particles and their next collision times with their two nearest neighbors. Also, by using a *heap* structure [5,6,14], we can find the minimum of the set of collision times using $\mathcal{O}(\log N)$ operations per collision. Although known since a rather long time, it is only recently that the heap concept has been used in physical problems, like front propagation [15] or molecular dynamics simulations of hard-sphere systems [8,9,12]. In this paper, we extend this technique to systems with force fields, provided these are Lagrangian invariant, or quasi-invariant.

There are still many open questions regarding ergodicity or the nature of equilibrium of self-gravitational systems, especially in one dimension [10,16,17]. It is thus very important to be able to simulate systems with a large number of particles for long times. We have tried to optimize as much as possible the speed of our heap implementation, as well as the numerical accuracy of the computation of particle tra-

* Corresponding author.
E-mail addresses: anz@obs-nice.fr (A. Noullez), fanelli@nada.kth.se (D. Fanelli), eaurell@nada.kth.se (E. Aurell).

0021-9991/03/$ - see front matter
doi:10.1016/S0021-9991(03)00048-2

jectories, to avoid apparently chaotic behavior coming in fact from roundoff errors. The paper is organized as follows. In Section 2, we discuss how the base concept of a heap can be improved for speed. Section 3 then shows how an efficient event-driven evolution scheme can be implemented by using the heap structure. In Section 4, we apply this code to the numerical solution of the one-dimensional self-gravitating system. This section also contains checks on the speed of the algorithm. In Section 5, we sum up our results and discuss future applications.

2. Heap optimizations

The base idea of heaps [5,6,14] is to put key elements in a binary tree and ensure that they satisfy the *heap condition*, that is that the value in any tree node is smaller than the value in its child nodes. This does not completely order the set, but is enough to warrant that the smallest value in the heap is at the root. Also, the heap condition can be maintained efficiently: if a node value is modified so that the heap condition is violated, we exchange the value with its parent node (if the value decreased) or with the smallest of its child nodes (if the value increased) and we repeat the procedure, moving up or down the tree until the heap condition is satisfied again or we reach the root or the leaves of the tree. The maximum number of operations is bounded by the height of the tree, that is $\log_2(N)$. Also, binary heaps can be represented efficiently as arrays with the children of node i being at locations $2i$ and $2i+1$, while its parent is $\lfloor i/2 \rfloor$.

The concept of trees, and thus of heaps, can be generalized to bases larger than two [7]. In a base r tree, each node has at most r subtrees so that the children of node i are $ri+2-r,\ldots,ri+1$ while the parent of node i is $\lfloor (i-2+r)/r \rfloor = \lceil (i-1)/r \rceil$. The height of the tree $h = \log_r N$ can be reduced by increasing r; on the other hand, the work needed at each level of the tree to find the smallest child increases linearly with r. Minimization of the expression $r \log_r N$ suggests that the best branching ratio should be e, the base of natural logarithms, but the processing of each level also incurs some work independent of r, and it is thus better to choose some higher (integer) value. This is especially important on modern microprocessors that access memory through *caches* which are filled in bursts of typically 4, 8, or 16 words. As the children of a node are stored consecutively in memory, they can also be loaded all at the same time when fetching a cache line, if the heap is *cache-aligned* [7], which can be realized by fiddling with the base address of the heap. On all microprocessors we used (Alpha, Pentium, MIPS), we found that base 4 was much better than base 2, while bases 8 and higher were slightly slower than base 4. The gain in speed between aligned base 4 heaps and unaligned base 2 ones is significant, giving a factor 2 speedup in the heap processing, with 60% coming from the choice of base and 40% from the memory alignment.

In our application, the elements have to be ordered both by collision times and by position, to find quickly the neighbors of a colliding pair. The standard solution to this problem is to use a single sorted instance of all elements, and to use *indirect* heap(s) containing only pointers to them [14]. But to get all the benefits of aligned large base heaps, the comparison keys have to be present in the heap and not accessed through pointers that would incur extra memory loads. We thus implemented *semi-indirect* heaps in which the keys are inside of the heap, along with the pointers to the corresponding elements. Because pointers have to be exchanged only when doing a swap, this implementation reduces nearly by half the number of memory accesses (to at most $r+1$ memory loads and three memory writes if a swap occurs in a sift-down) and is faster than other priority queue implementations like those decribed in [9].

3. The algorithm

We consider the motion of N colliding particles in a one-dimensional medium. The interaction is not specified at this level: we only require that the equation of motion for a particle can be integrated in between

two successive collisions. Arrays of size N contain the states of the particles, such as position, velocity, and acceleration, at the time of their last collision, stored in increasing order of the spatial coordinates. An additional state variable associated to each particle is τ_j, the time it last experienced a collision.

The algorithm starts by computing the collision time of each particle with its neighbor to the right, and the results are stored in an array of size $N-1$, which is then turned into a heap. So that we do not need to move the whole particles state while processing the heap, we introduce an indexing array, Particle-Heap ($PH[\bullet]$), mapping the position in the heap to the rank in space of the leftmost of the two particles (j and $j+1$) involved in that collision (see Fig. 1). To update the list of predicted collision times of neighbors particles, we also need the index array inverse to Particle-Heap, which we call Heap-Particle ($HP[\bullet]$). Hence for all j in the range 1 to $N-1$, $PH[HP[j]] = j$ and $HP[PH[j]] = j$. This condition will be preserved at all times while we update the heap. Note that the collison times are really directly present in the heap, and that the two indexing arrays then realize exactly the functions needed to implement the semi-indirect heap.

Once the heap has been built, the minimum collision time $t_{\min}$ is at the root. The particles involved in the first collision, $j = PH[1]$ and $j+1$, are selected and their states evolved up to $t_{\min}$ where they are rearranged by the collision (momenta simply exchanged in the case of elastic collision) and τ_j and τ_{j+1} are set equal to $t_{\min}$. Next the new predicted collision time between j and $j+1$ is computed and replaces the one at the root of the tree. The root might now not fulfill the heap condition, so the heap array is re-arranged by sifting down the root value, using at most $\mathcal{O}(\log N)$ operations, as discussed previously.

The next collision times of particles j and $j+1$ with their other nearest neighbor, $j-1$ and $j+2$, respectively, also need to be re-computed, see Fig. 2. To do this, particles $j-1$ and $j+2$ are temporarily moved forward in time up to $t_{\min}$, where their new collision times are computed and put into the heap at $HP[j-1]$ and $HP[j+1]$, replacing the old ones. As a consequence, the heap has to be re-arranged two more times, again at a cost of at most $\mathcal{O}(\log N)$ for each modification.

The heap is now again in a consistent state with the next collision time at the root, and the whole procedure can be repeated. The evolution can be stopped either after some fixed number of collisions Z, or when the predicted time for the next collision becomes larger than some chosen final time T_{end}. In the end, all particles are moved forward in time from their own τ_j to the final time which is either T_{end} or the time of the last collision. In conclusion, the complexity of the algorithm is in the worst-case $\mathcal{O}(Z \log N)$ plus lower-order terms $\mathcal{O}(Z)$ and $\mathcal{O}(N)$.

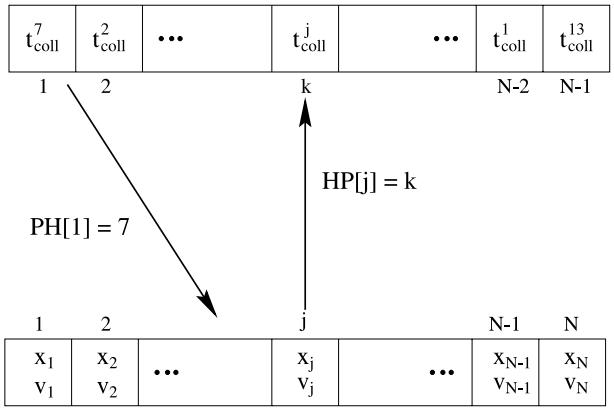

Fig. 1. This figure shows the structure of a semi-indirect heap and the function of the two "shuffling" arrays $PH[\bullet]$ and $HP[\bullet]$. The first array in the figure only contains the predicted collision times ordered as a heap, while the second contains the particle states stored in increasing order of spatial positions. The two indexing arrays allow to move back and forth between the two sets.

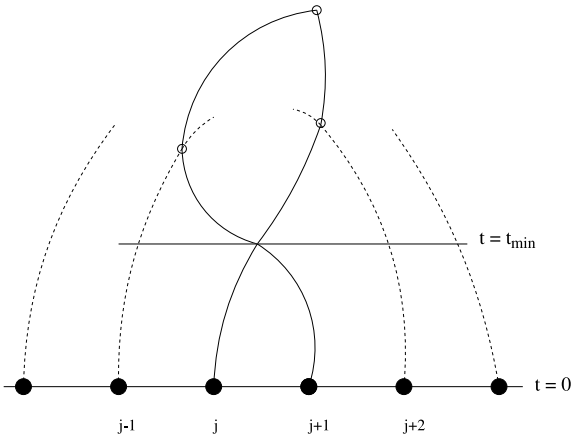

Fig. 2. Intersection of the trajectories of particles j and $j+1$ at time $t = t_{\min}$. The ringed intersections are the collision/crossings that need to be recomputed.

4. Applications

Consider a one-dimensional (classical) self-gravitating system of N particles, all of the same mass m normalized to be N^{-1}. The Hamiltonian is

$$H = \sum_{j=1}^{N} \frac{p_j^2}{2m} + 2\pi G m^2 \sum_{j=1}^{N} \sum_{i>j}^{N} |x_i - x_j|, \tag{1}$$

where x_j is the position of particle j, $p_j \equiv m v_j$ is the momentum conjugate to x_j, and G is the gravitational constant [3,4,10,16,17]. We choose as unit of length the spatial interval in which the particles are initially contained, so the initial density ρ_0 is equal to one. Also, the natural time scale which is the inverse of the Jeans frequency $\omega_{\mathrm{J}} = (4\pi G\rho_0)^{1/2}$ can be set to unity if we take $4\pi G$ equal to one. Inbetween two collisions, the acceleration of each particle is constant, and is proportional to the difference of number of particles on its right and on its left, so that in the (x,t) plane, the path of particles between collisions follows a parabola.

In this problem, the time evolution calculations have to be done with the greatest care to reduce numerical errors. Indeed, the system is chaotic, i.e., dynamically unstable, and amplifies small perturbations. First, finding the collision time of two particles implies solving a quadratic equation, which should be done using the stable form of the schoolbook formula [11]. Next, moving the particles forward in time involves evaluating a second-order polynomial and must be done using *Horner's rule* to increase accuracy and reduce the number of floating-point operations to two on modern microprocessors equipped with a *fused* multiply–add operation. Also, the collision point of two particles must be done using a common symmetric formula to ensure they end up at the same point without systematic drifts.

Fig. 3 shows phase space portraits of this self-gravitating dynamics with particles initially uniformly distributed in space, and velocity a smooth function of position. After caustic formation, the system develops a spiral structure in phase-space.

The performance of the algorithm is confirmed by Fig. 4, which shows CPU time per collision vs. number of particles in semi-logarithmic scale. The linear dependence on $\log N$ is clear in the range 300–10000, and there is also a significant constant contribution coming from the floating point operations needed to update the particles states. In the data of Fig. 4 (see caption), we reached speeds in excess of 4×10^5 collisions/s, while on a DEC ALPHA-based workstation at 600 MHz, we got speeds of around

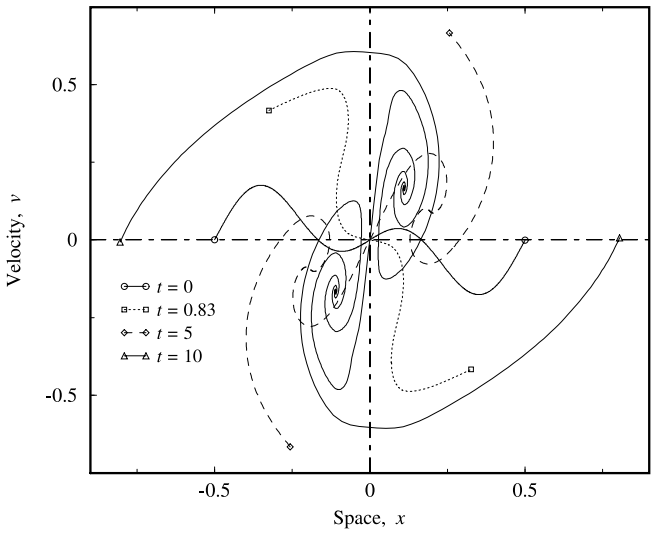

Fig. 3. Phase space portrait of a self-gravitating system. The initial velocity profile is a double sine wave. In a short time, a caustic is formed, where velocity is a multi-valued function of position and, after a few Jeans times, the system develops spiral structures in phase space, due to the gravitational attraction preventing the particles from running away.

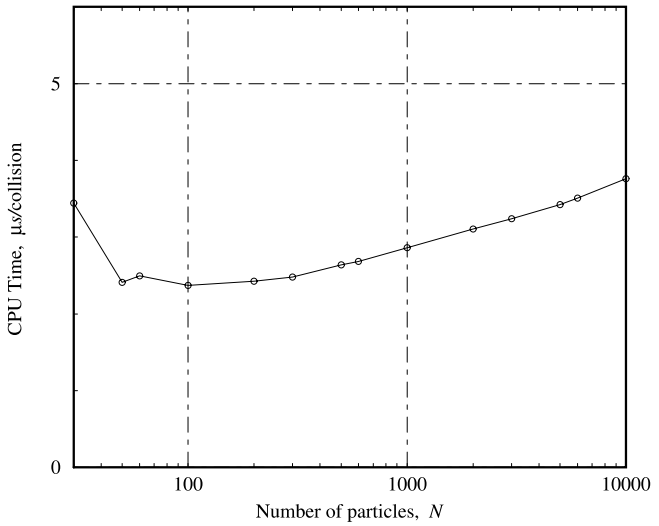

Fig. 4. CPU time per collision measured as the output of the UNIX library function `times( )` divided by the number of collisions, after 30 Jeans time, for systems of various size N. The code was compiled by `gcc` on the Linux kernel 2.2 and ran on an Intel Pentium II 450 MHz processor. Data points on the left are not very reliable because of the limited resolution of `times( )`.

1.3×10^6 collisions/s for N equal to 1000. Hence this algorithm allows the study of fairly large systems for long times on ubiquitous hardware.

The preceding discussion has made clear that the limiting factor for simulating a system up to a given time T_{end} is the number of collisions Z that occur in that time interval. It is thus especially important to know how Z scales with N. Heuristically, if the velocity of the particles is independent of N and their

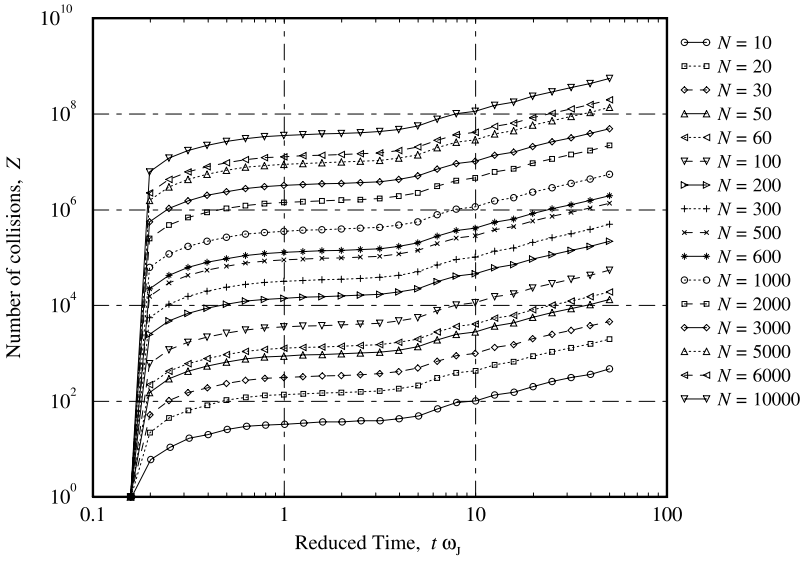

Fig. 5. Number of collisions vs. time for different number of particles. We can clearly distinguish an early regime, before many collisions have occurred, a late regime (after about $10\omega_J^{-1}$) where the collision rate becomes constant, and an intermediate regime. For all times, the number of collisions goes as N^2.

separation is $\approx N^{-1}$, one expects that the average time between successive collisions for a given particle goes like N^{-1}, so the total collision rate should grow as N^2. Fig. 5 shows the number of collisions vs. physical time, in double logarithmic coordinates, for different number of particles. Curves corresponding to different Ns are parallel to each other and separated by the square of the ratio of their respective N, showing that the proposed scaling holds true for different discretizations of the same initial conditions for all regimes.

5. Conclusions

We discussed the implementation of a fast heap-based event-driven scheme for integrating numerically one-dimensional systems of N interacting particles, provided the dynamics can be integrated between two successive collisions. The collision times are ordered on a heap, reducing the complexity to $\mathcal{O}(\log N)$ operations per collision. As a consequence, for large values of N, the present algorithm is faster than earlier algorithms in the litterature, which are $\mathcal{O}(N)$. This opens up the perspective of studying the statistical mechanics of such systems for large number of particles and long times.

In the paper, we presented classical (Newtonian) self-gravitating systems as one possible application of our algorithm. Nevertheless, it is worth stressing that the algorithm is more general and, for example, can also be applied to models of the motion of matter in an expanding Universe [1,2,13].

Acknowledgements

We thank U. Frisch, M. Hénon, and P. Muratore-Ginanneschi for discussions. We also thank D. Zanette for pointing out reference [9]. This work was supported by RFBR-INTAS 95-IN-RU-0723 (E.A. and D.F.), by the Swedish Natural Science Research Council through Grants M-AA/FU/MA 01778-333 (E.A.) and M-AA/FU/MA 01778-334 (D.F.).

References

[1] E. Aurell, D. Fanelli, P. Muratore-Ginanneschi, Physica D 148 (2001) 272.
[2] D. Fanelli, E. Aurell, Astron. Astrophys. 395 (2002) 399.
[3] J.M. Dawson, Phys. Fluids 5 (1962) 445.
[4] O.C. Eldridge, M. Feix, Phys. Fluids 5 (1962) 1076.
[5] J.H. Kingston, Algorithms and Data Structures—Design, Correctness, Analysis, Addison-Wesley, Reading, MA, 1998.
[6] D.E. Knuth, The Art of Computer Programming, Volume 3 Sorting and Searching, Addison-Wesley, Reading, MA, 1973.
[7] A. LaMarca, R.E. Ladner, ACM J. Exp. Algorithmics 1 (1996) 4.
[8] M. Marin, D. Risso, P. Cordero, J. Comput. Phys. 109 (1993) 306.
[9] M. Marin, P. Cordero, Comput. Phys. Commun. 92 (1995) 214.
[10] C.J. Reidl, B.N. Miller, Phys. Rev. A 46 (1992) 837;
Phys. Rev. E 48 (1993) 4250;
B.N. Miller, P. Youngkins, Phys. Rev. Lett. 81 (1998) 4794.
[11] W.H. Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling, Numerical Recipes in C: The Art of Scientific Computing, Cambridge University Press, Cambridge, 1988.
[12] D.C. Rapaport, J. Comput. Phys. 34 (1980) 184.
[13] J.L. Rouet, M.R. Feix, M. Navet, Vistas in Astronomy 33 (1990) 357;
J.L. Rouet et al., Lecture Notes in Physics: Applying Fractals in Astronomy, vol. 161, 1991.
[14] R. Sedgewick, Algorithms in C, Addison-Wesley, Reading, MA, 1990.
[15] J.A. Sethian, SIAM Rev. 41–42 (1999) 199.
[16] G. Severne, M. Luwel, Astrophys. Space Sci. 122 (1986) 299.
[17] T. Tsuchiya, T. Konishi, N. Gouda, Phys. Rev. E 50 (1994) 2607;
T. Tsuchiya, N. Gouda, T. Konishi, Phys. Rev. E 53 (1996) 2210.