Available online at www.sciencedirect.com
ScienceDirect
Journal of Computational Physics 225 (2007) 829–839
JOURNAL OF COMPUTATIONAL PHYSICS
www.elsevier.com/locate/jcp

Optimization of PIC codes by improved memory management

D. Tskhakaya [a,*], R. Schneider [b]

[a] Department of Theoretical Physics, Association-Euratom ÖAW, University of Innsbruck, A-6020 Innsbruck, Austria
[b] Max Planck Institute für Plasmaphysik, EURATOM Association, Wendelsteinstrasse 1, D-1749 Greifswald, Germany

Received 15 March 2006; received in revised form 18 December 2006; accepted 4 January 2007
Available online 13 January 2007

Abstract

A simple method is described for optimization of particle-in-cell codes by improved memory management. This includes a faster calculation of Monte-Carlo collision operators. It is demonstrated that the CPU time can be reduced by a factor of 2 and more without reduction of the simulation accuracy.

PACS: 52.65.Rr; 52.65−y

Keywords: PIC; Monte Carlo; Binary collision

1. Introduction

Particle-in-cell (PIC) codes are extensively used for different plasma studies. Large number of simulation particles (10^5–10^{10}) and spatial grid cells (10^2–10^7) used in PIC simulations [1] make the corresponding runs extremely time consuming. Some simulation runs can take 10^4 CPU hours. As a result, the optimization of PIC codes is an important and necessary task for increasing of capabilities of corresponding simulations [2].

Originally PIC codes have been developed for simulations of collisionless plasma (see [3]) and corresponding code optimizations deal with the particle mover, the solver of the Maxwell equations and parallelization of these codes (e.g. see [4,5]). Later, different kinds of particle collisions have been added to the PIC codes, so that most of them represent hybrid PIC + Monte Carlo (MC) codes simulating a collisional plasma [6–12]. Here, under "PIC" and "MC" we mean the parts of the code simulating collisionless plasma and particle collisions, respectively. The MC part in PIC simulations can be as (or even more) time consuming as the PIC part, so that only the optimization of the PIC part is no longer sufficient.

In the present work, we describe one possible optimization, which can be easily implemented in the PIC codes and reduce the CPU time (for a single processor) up to 50%. The main idea is very simple: Simulation particle information is stored in the memory not randomly anymore, but according to the spatial grid cell. In a

* Corresponding author. Permanent address: Institute of Physics, Georgian Academy of Sciences, Tbilisi, 380066, Georgia.
E-mail address: david.tskhakaya@uibk.ac.at (D. Tskhakaya).

0021-9991/$ - see front matter
doi:10.1016/j.jcp.2007.01.002

conventional code neighboring (in memory) particles i and $i+1$ can be in different cells, while in this new memory storage concept they are in the same one. The optimization has been done for the 1D3V (one spatial and three velocity dimensions) electrostatic "single processor" code BIT1 [13], which was developed on the basis of the XPDP1 code [14]. Generalization for higher dimensional, paralleled and electromagnetic codes is trivial. The BIT1 (and other codes from XPDP code family) are written in C. Nevertheless, we expect that similar results can be obtained using other programming languages (Fortran, C++, etc.).

We note that the idea of particle sorting according to the grid cells is not new and comes from the PIC codes incorporating particle–particle collisions. The particles inside a collision cell were initially grouped via linked lists (e.g. see [6]). But later it has been recognized that it is better to use traditional particle arrays and apply a special routine for particle sorting according to the cells after a given number of time steps [5,15–18]. Moreover, it has been demonstrated that this sorting can significantly increase the simulation speed of PIC even without collisions [5,15,17]. In the next sections, we describe what is new in our approach and which advantages can be expected.

Originally, the optimization has been done to reduce the CPU time for Coulomb collisions, but as test runs show it has been significantly reduced for the PIC part too. For convenience, we consider optimization for each part of the code separately.

The paper is organized as follows. In Sections 2 and 3, we consider optimization of the PIC and MC parts, respectively. The results are summarized in Section 4.

2. Optimization of the PIC part

In PIC codes the position of particle in simulation area and computer memory are decoupled. Hence, the neighbouring particles (in computer memory) can be far away from each other in the simulation area and suffer different forces and even different physics. Hence, running over particle index the routines like particle mover or charge assignment have to operate with different external variables, e.g. fields and densities at different grid points. As a result, the probability that this values will be found in a fast (cache) memory can be low. This can be avoided if particles are sorted according to the grid cells. In this case operations on neighbouring particles require the same external variables and the probability of a "cache-hit" is higher. An obvious way to reach this is to sort particles before applying the particle mover and weighting operations. In [5] it has been demonstrated that using an usual sorting algorithm one can speed-up simulations up to 30%. A more optimized algorithm has been proposed in [17], where the particle array is passed just two times during the sorting. Correspondingly, the speed-up can reach 40–70% (the exact number depends on simulation details and processor type). An interesting approach has been developed in [18]. Here, particles are sorted according to the spatial cells like in the approach presented here, but in addition, indices of the first particle in each cell and the corresponding number of particles in the cell are also saved. This probably speeds up the next sorting and can be directly used for particle–particle collisions inside the cells. Although, this approach is very attractive and requires fewer efforts for modification of the conventional PIC codes, it is not very flexible for codes incorporating inelastic particle collisions (or other processes) with sources and sinks. The problem is that by creating or deleting particles the sorting can be lost and an additional one can be necessary. The number of calls of this sorting algorithm could increase with increasing number of inelastic processes described above. Taking into account that sorting is a time-consuming operation, we realise that the sorting method described before is not optimal for the complicated collision operators.

In the present work, we propose a new method of memory management, which allows keeping of the particle sorting through the whole simulation. As a result, no additional sorting is required for increasing number of implemented collision processes.

In a conventional PIC code the data of ith particle are saved in the computer memory as an array $A[s][i]$, where A can be particle position, or velocity components and s denotes the particle type, e.g., $s=0$ and $s=1$ can correspond to the electrons and ions, respectively. One can save the same data in the following format:

$$A'[s][k][i]. \tag{1}$$

Here, k denotes the (spatial) cell number where this particle is located at a given time. For the particle position, A' represents the relative position inside the k cell, otherwise A' represents the same value as A. For N-dimen-

sional case k represents a N-dimensional (integer) vector. As a result, the particles neighboring in space are also neighbors in the memory and are sorted according to the spatial cells (see Fig. 1). Let us estimate the advantages of this new format:

1. In a conventional case particle positions (x) are (usually) saved as a double float variable. This high accuracy is required when there is a large difference between the system size, L, and the distance gained by particles each time step (Δt), $\delta x = V\Delta t$. If $\delta x/L \lesssim 10^{-6}$, then the float accuracy cannot be enough for calculation of the particle position near the end of the system. When the format (1) is used the maximum distance associated with the particle corresponds to the cell size. Hence, particle positions can be saved as float variables with higher precision. Operations on a double float are effectively slower than ones on single float. There are two reasons for this: First of all, some processors perform operation on single float faster than on double float, and second, due to reducing of the required memory more particle and field data can be stored in cache memory, hence increasing the probability of a "cache-hit" and the simulation speed. The corresponding gain in the CPU time (per time step) can be given as $DN_p t_f$, where D is the (spatial) dimensionality of the code, N_p is the number of simulation particles and t_f is the average gain in time due to operations on a single particle (per dimension). t_f can be assumed to be independent of N_p.

It has to be taken into account that this transform from double to single float variables will reduce the required memory, which is an important factor when writhing the given state of large size systems to disk space.

2. In a conventional code, one has to calculate during particle and force weighting (i.e. calculating the density associated with the given particle and the force acting on it) in which cell, k, the particle is located and its the relative position inside the cell. Thus, the following operations (one conversion to integer and one arithmetical one) have to be performed per spatial dimension:

$$k = (int)x[s][i]; \tag{2}$$

$$\delta x = x[s][i] - k; \tag{3}$$

Here and below (int) denotes the integer part of the number. In this new approach, these values are known in advance. Assuming that each of this operations requires the same time, t_a, we obtain the corresponding gain in the CPU time as $4DN_p t_a$. We have to keep in mind that the time required for performing these operations depends on the type of given processor and the introduced assumption can be used just for rough estimates.

3. A significant reduction of simulation run-time can be obtained due to increased cache-hit probability for particle and field arrays, which was described above. The corresponding gain in the CPU time, t_m, depends strongly on the used processor type (see [5,17]), on the structure of the given code and simulated model.

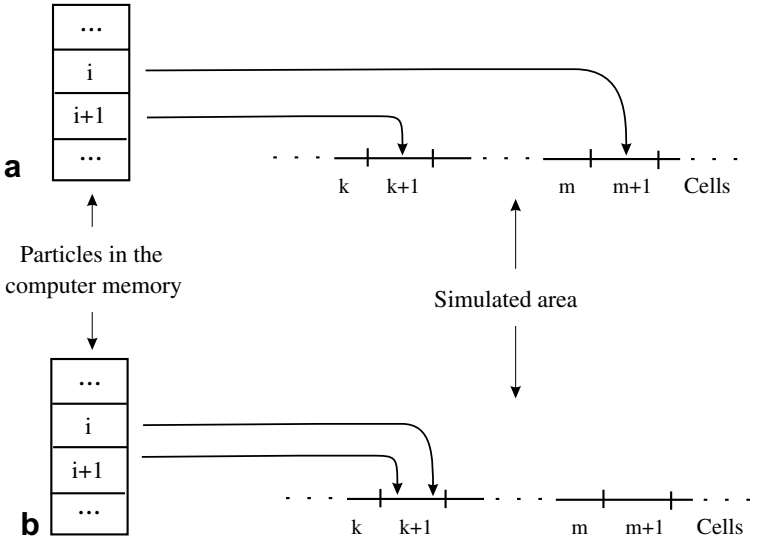

Fig. 1. Representation of two neighboring particles in a conventional (a) and the new (b) PIC code.

On the other hand, the proposed approach requires an additional CPU time: each time step some particles cross the boundary between neighboring cells and have to be rearranged in memory. This arrangement consists of two steps: (i) Checking of new position of particles, and (ii) if a given particle left its cell, changing correspondingly its address. The first step requires no additional time, because in conventional PIC codes the check of new positions is anyway needed in order to identify particles crossing the boundary of the simulated area. The second step for each particle crossing a cell boundary requires an additional time $2(1+D+V)t_a$, where V is the dimensionality in velocity space. Here, we assume again that each operation requires a time t_a. For example, if the particle i jumped from the k to the m cell, then the corresponding arrangement for the 1D1V code can be done in the following way:

Moving the address of the particle i into the new place in the array:

```
np[s][m] +=1;
x[s][m][np[m]] = x[s][k][i];
v[s][m][np[m]] = v[s][k][i];
```

Removing of i particles address from the old place in the array:

```
np[s][k] -=1;
x[s][k][i] = x[s][k][np[k]];
v[s][k][i] = v[s][k][np[k]];
```

Here, $np[s][p]$ represents the number of particles of the type s in the cell p. The total increase of the CPU time, which is required for this arrangement per time step can be given as follows: $2(1+D+V)\alpha N_p t_a$. Here, α represents the fraction of particles crossing a cell boundary per time step. In general, α depends on number of cells, their size, particle distribution and the time step, but independent of N_p. Finalizing our estimates we obtain the following difference in CPU time, which is required for simulation of one time step for conventional and new codes:

$$\delta t = N_p(2bt_a - Dt_f) - t_m, \quad b = \alpha(1+D+V) - 2D. \tag{4}$$

α and b can be estimated in the following way: The density of particles, which are at the point $\vec{r}$ in the grid cell k and cannot leave this cell after one time step, is given as

$$N_k^{nl}(\vec{r}) = \int_{-\frac{\Delta\vec{r}}{\Delta t}}^{\frac{\Delta\vec{r}-\vec{r}}{\Delta t}} f_k(\vec{r},\vec{v})\,\mathrm{d}\vec{v}, \tag{5}$$

where $\Delta\vec{r} = (\Delta x, \Delta y, \Delta z)$ is the size of the cell and f_k is the particle distribution function in this cell. The total number of particles leaving this cell after one time step will be

$$N_k^l = N_k - \int_0^{\Delta\vec{r}} N_k^{nl}(\vec{r})\,\mathrm{d}\vec{r}, \tag{6}$$

where N_k is the total number of particles in the k cell. After changing the integration order and neglecting spatial gradients inside the cell we obtain

$$\alpha_k = \frac{N_k^l}{N_k} \approx 1 - \int_{-\frac{\Delta\vec{r}}{\Delta t}}^{\frac{\Delta\vec{r}}{\Delta t}} \left(1 - \frac{|v_x|\Delta t}{\Delta x}\right)\left(1 - \frac{|v_y|\Delta t}{\Delta y}\right)\left(1 - \frac{|v_z|\Delta t}{\Delta z}\right) F_k(\vec{v})\,\mathrm{d}\vec{v}, \tag{7}$$

$$F_k(\vec{v}) = \frac{\int_0^{\Delta\vec{r}} f_k(\vec{r},\vec{v})\,\mathrm{d}\vec{r}}{N_k}. \tag{8}$$

The 1D and 2D cases correspond to $\Delta y = \Delta z \to \infty$ and $\Delta z \to \infty$, respectively. Taking into account the Courant condition for the explicit codes [3,6], $\Delta r/\Delta t > V_{max}$, where V_{max} is the maximal velocity of simulated particles, we obtain

$$\alpha_k \approx 1 - \left(1 - \frac{u_{k,x}\Delta t}{\Delta x}\right)\left(1 - \frac{u_{k,y}\Delta t}{\Delta y}\right)\left(1 - \frac{u_{k,z}\Delta t}{\Delta z}\right), \tag{9}$$

$$b \approx 1 - D + V - (1 + D + V)\left\langle \left(1 - \frac{u_x\Delta t}{\Delta x}\right)\left(1 - \frac{u_y\Delta t}{\Delta y}\right)\left(1 - \frac{u_z\Delta t}{\Delta z}\right)\right\rangle. \tag{10}$$

Here $\vec{u}_k$ is the characteristic velocity of particles in the k cell and the brackets $\langle\rangle$ denote averaging over all grid cells $\langle A\rangle \equiv \sum N_k A_k / N_p$. As we see, the sign of b can depend on the ratio $u\Delta t/\Delta r$. Usually in PIC simulations one uses $\Delta r/\Delta t \gg u$, so that $b < 0$ and $\delta t < 0$ for any values of t_a, t_f and t_m. Thus, using the new approach we reduce the CPU time. Moreover, the simulations described below show that the contribution of the first term in expression (4) is negligibly small and the CPU time reduces almost independently of the ratio $u\Delta t/\Delta r$.

In Fig. 2 the relative CPU time, t_{new}/t_{con}, is plotted versus simulation parameter $\Delta x/V_T\Delta t$ and b. Here, t_{new} and t_{con} are the CPU times needed for the new and conventional codes, respectively, and V_T is the electron thermal velocity. The simulations correspond to an unbounded collisionless Maxwellian plasma without magnetic field. In the simulations all parameters, except Δx, were fixed. These results indicate that the gain in the CPU time can be over 50% and depends weakly on b.

We note that all simulations in this work have been performed by conventional and updated (according to (1)) versions of the 1D3V electrostatic code BIT1 [13]. The simulations were running on a PC under Linux Platform.

During the simulation the number of particles in each grid cell oscillates. Hence, in order to ensure that the size of particle arrays is large enough we use $I = (1.5 \div 2)N_{max}$, where I is the size of the array corresponding to the index "i" in (1) and N_{max} is the maximum expected number of particles per grid cell. This number has to be kept as small as possible, otherwise there will be no overall gain in the reduction of the size of the particle array (see point 1). We note that in our case the simulation speed was insensitive to the exact choice of factor $1.5 \div 2$.

In order to investigate the dependence of the gain in CPU time on different simulation parameters we have performed additional test runs. As before, the simulations correspond to an unbounded collisionless Maxwellian plasma without magnetic field. The plasma density and time step were fixed, $\Delta t = 0.05/\omega_p$. The simulated number of cells and average number of particles per cell cover practically the whole range of interest. The results are plotted in Fig. 3 indicating significant reduction of the CPU time. As it was expected from Eq. (4), the relative CPU time decreases linearly with increasing number of simulated particles

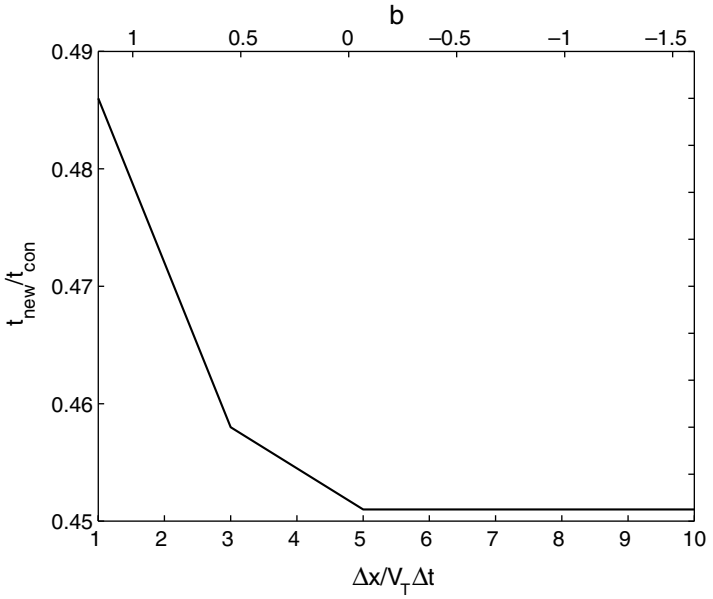

Fig. 2. Relative CPU time, t_{new}/t_{con}, versus simulation parameter $\Delta x/V_T\Delta t$. Following simulation parameters have been used: Number of cells = 100, number of particles per cell = 250, $\Delta t\omega_p = 0.2$. ω_p is the plasma frequency.

for the fixed number of cells. When the number of cells increases for a fixed number of particles per cell, then the CPU time spend for the field solver increases. This part is the same in conventional and new codes and needs the same amount of the CPU time, so that the relative CPU time slightly increases. We note that results, e.g., the particle distributions, plasma profiles and oscillation spectra, from the conventional and new codes are identical.

3. Optimization of the MC part

During the last decades different collision models have been developed for PIC simulations. In this work, we describe a new optimized model for Coulomb collisions and assume that collided particles have the same weight. Generalization for the multi-weight case will be given elsewhere.

The Coulomb collision operators used in PIC codes can be divided into two categories. The operators of the first type calculate the Rosenbluth potentials and correspondingly, the average force acting on particles due to the Coulomb collisions [19]. In order to achieve sufficient accuracy in calculation of the Rosenbluth potentials, an extremely large number of simulation particles is required [20], making it practically impossible to simulate large systems. As a result, in practice only linear operators of this type are used (e.g. [21]). The operators of the second type represent nonlinear operators and are based on the "binary collision" model introduced in [7]. These operators use the following fact: The typical size of the spatial cell in PIC simulations is of the order of a Debye radius, λ_D. Since, there is no need to consider the Coulomb collision between two particles separated by a distance larger than λ_D, the interaction between the particles in different cells can be neglected. As a result, one can consider collision of particles only inside the PIC cells.

The binary collision model consists of three steps: (i) First, particles are arranged in cells, (ii) then these particles are paired, so that one particle has only one partner and (iii) paired particles are (statistically) collided (see Fig. 4). The scattering probability in the center-of-mass system is calculated according to the following expression:

$$P(\chi) = \frac{\chi}{\langle\chi^2\rangle_t}\exp\left(-\frac{\chi^2}{2\langle\chi^2\rangle_t}\right), \quad \langle\chi^2\rangle_t \equiv \frac{e_1^2 e_2^2}{2\pi\varepsilon_0^2}\frac{n\Delta t_c \Lambda}{\mu^2 V^3}. \tag{11}$$

Here, $e_{1,2}$, $\mu = m_1 m_2/(m_{1+}m_2)$ and V denote the charge, reduced mass and relative velocity of the collided particles, respectively. n, ε_0 and Λ are the density, vacuum permittivity and Landau logarithm, respectively. Δt_c is

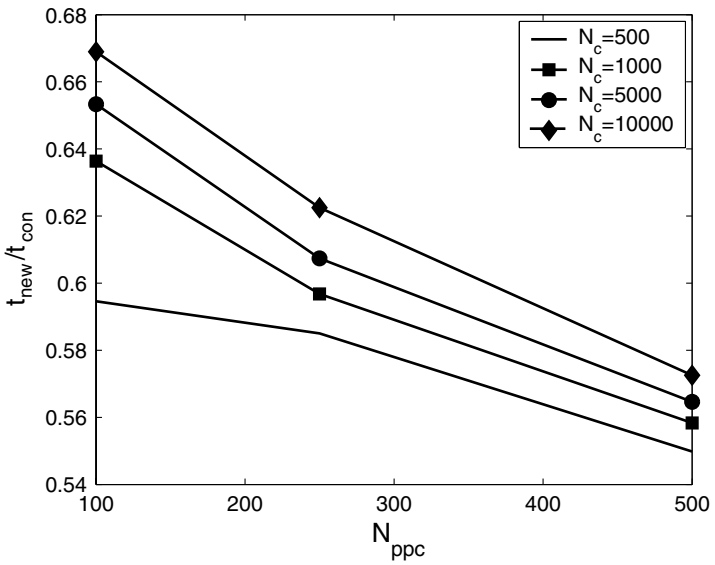

Fig. 3. Relative CPU time versus number of particles per cell, N_{ppc}, for different number of cells, N_c. The plasma density and time step are fixed, $\Delta t = 0.05/\omega_p$.

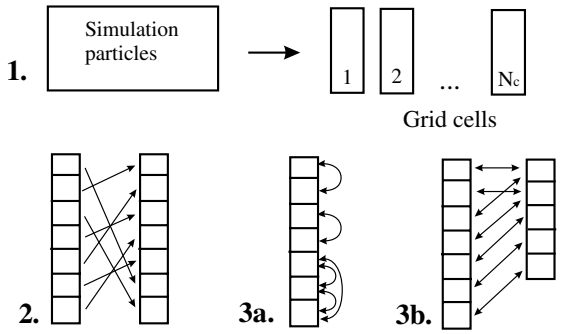

Fig. 4. Binary collision model from [7]. (1) Grouping of particles in the cells. (2) Randomly changing the particle order inside the cells. (3a) Colliding particles of the same type. (3b) Colliding particles of different types.

the collision time step. This operator is sufficiently fast: each particle collides only with one other particle, i.e. the operator scales as $N_p/2$. It is very accurate: energy and momentum are conserved and the typical plasma relaxation times are reproduced within the accuracy of a few percent. Δt_c in general differs from Δt ($\Delta t_c > \Delta t$) and should satisfy the condition

$$v_c \Delta t_c \ll 1, \tag{12}$$

where v_c is the Coulomb collision frequency.

There exist further modifications of this original binary collision model. The operator introduced in [10] collides all particles inside the cells. As a result, this model reproduces slightly more accurate relaxation times, but for the price of a significant larger CPU time. It scales as $N_p^2/2N_c \gg N_p/2$. Another model proposed in [11] uses a "cumulating scattering": In spite of making a large number of small scattering angle collisions, particles suffer a single large angle collision. This allows to avoid the condition (12) and significantly reduce the corresponding CPU time. On the other hand, calculation of the corresponding cumulative χ requires additional conditions, e.g., for a strongly nonuniform plasma this model probably fails. We note that in all described operators the first step represents grouping of particles according to the grid cells. Using the memory allocation method (1) particles are ultimately grouped according to the cells, so that no additional grouping is required. Hence, a significantly amount of the CPU time can be saved.

Below, we describe a new binary collision model, which was introduced in the BIT1 code. The aim was to develop a as fast as possible collision operator scheme without any significant reduction of accuracy.

The model is based on the original operator from [7] and consists of two steps: (i) Choosing of colliding particle pairs inside cells, and (ii) collision of paired particles. The later represents angular scattering according to (11) (and identical to one from [7]). The first step differs from the corresponding one from [7] and will be considered in detail. In the new operator, the collided pairs are randomly picked up from the cell, so that no "randomization" of particle addresses is done (step 2 in Fig. 4). Moreover, using the fact that particle collisions in different cells are statistically independent, in the new operator the same random numbers are used for different cells. This saves additionally a significant amount of the CPU time. This approach, allowing for particles multiple (or zero) collisions during the Δt_c, exactly corresponds to the *statistical* meaning of the scattering angle from (11) (see [7,22]): An average particle should collide not *necessarily* once, but in *average* once per time interval Δt_c. The corresponding scheme for generating of colliding particle pairs is shown in Fig. 5. When colliding particles of the same type, first N_m random numbers are generated and saved. Here, N_m is the maximum number of particles in the cells. Then, for each cell two random numbers, e.g. r and f are taken and particles $i = (int) r N_k$ and $j = (int) f N_k$ are collided. N_k is the number of particles in the given k cell. This process is continued until all $N_k/2$ collisions are completed in the k cell. The same procedure takes place for the $k+1$ cell, and so on, until all cells are passed. It can happen that at some cells $i = j$. In this case, the collision partner is chosen to be $i+1$, or $i-1$. If N_k is an odd number then one particle is not collided.

Collisions of particles of different types 1 and 2 are performed in a similar way. First, $2N_m$ random numbers are generated and saved, where now N_m is the maximum number of particles in the cells, which have lower density. For simplicity, let us assume that in the given cell k particles of the type 1 have lower density. Then,

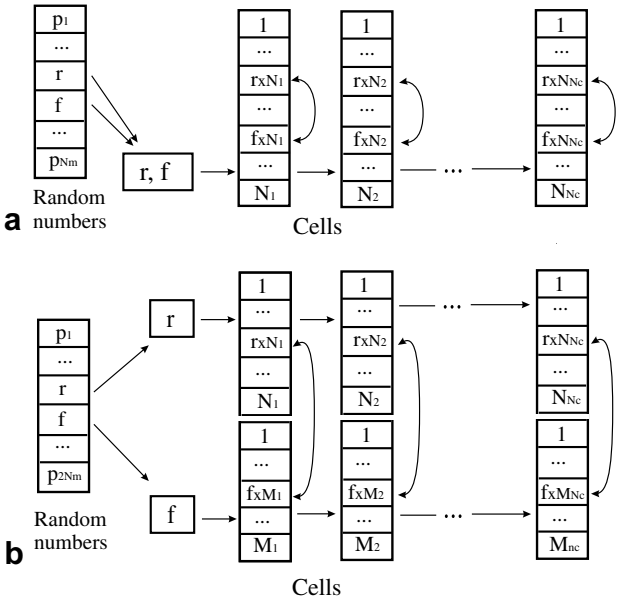

Fig. 5. Collision model from the new BIT1 code. (a) Collision of particles of the same type. (b) Collision of particles of different types.

for each collision one pair of saved random numbers, r and f, is used and particles $i = (int)rN_{1,k}$ of the type 1 and $j = (int)fN_{2,k}$ of the type 2 are collided. Here, $N_{1,k}$ and $N_{2,k}$ represent number of particles of the type 1 and 2 inside the given k cell. This process is continued until all $N_{1,k}$ collisions are performed in the k cell. The same procedure will take place for the other cells. The density in (11) corresponds to the particles with higher density, e.g., for the above considered k cell $n = n_2$.

Validity of the proposed operator has been checked via simulation of following relaxation processes [23]:

1. Slowing down of the cold electron beam due to collisions with cold ions:

$$\frac{\partial}{\partial t}\vec{V}_{\mathrm{e}} = -\nu_{\mathrm{sd}}\vec{V}_{\mathrm{e}}, \quad \nu_{\mathrm{sd}} = \frac{e_{\mathrm{e}}^2 e_{\mathrm{i}}^2}{\varepsilon_0^2}\frac{n\Lambda}{m_{\mathrm{e}}^2 V_{\mathrm{e},0}^3}\left(1+\frac{m_{\mathrm{e}}}{M_{\mathrm{i}}}\right). \tag{13}$$

2. Temperature isotropization of electrons due to self-collisions:

$$\frac{\partial}{\partial t}(T_{\parallel} - T_{\perp}) = -\nu_{\mathrm{ti}}(T_{\parallel} - T_{\perp}), \tag{14}$$

$$\nu_{\mathrm{ti}} = \frac{3e_{\mathrm{e}}^4}{8(\pi)^{3/2}\varepsilon_0^2}\frac{n\Lambda}{\sqrt{m_{\mathrm{e}}}T_{\parallel,0}^{3/2}}\frac{-3+(3-A)\tanh^{-1}\left(\sqrt{A}\right)/\sqrt{A}}{A^2}, \tag{15}$$

$$A = 1 - \frac{T_{\perp,0}}{T_{\parallel,0}} > 0. \tag{16}$$

3. Thermal equilibration of the plasma consisting of two Maxwell-distributed ion species with different temperatures:

$$\frac{\partial}{\partial t}T_1 = \nu_{\mathrm{te}}^{1/2}(T_2 - T_1), \tag{17}$$

$$\nu_{\mathrm{te}}^{1/2} = \frac{e_1^2 e_2^2}{3\sqrt{2}(\pi)^{3/2}\varepsilon_0^2}\frac{\sqrt{M_1}}{M_2}\frac{n_2\Lambda}{(T_1 + M_1 T_2/M_2)^{3/2}}. \tag{18}$$

Here, the subscript "0" denotes the initial value of corresponding parameter. Two sets of runs have been performed, one with the conventional binary collision model and another with the new collision operator. The field solver was switched off. The number of particles per cell was 100. Time steps satisfy the condition $\Delta t_c = \Delta t \ll (v_{sd,ti,te})^{-1}$. Simulation results are plotted in Figs. 6–8. As one can see, the difference between the results of the different code versions are well below the statistical fluctuation level (Figs. 7 and 8). The slight deviation of the simulation and analytic results for the slowing-down process (Fig. 6) is caused by the fact that due to collisions the electron beam temperature increases and the linear approximation (13) is not valid any more. Similar results have been obtained for different number of particles per cell: 50, 250 and 500.

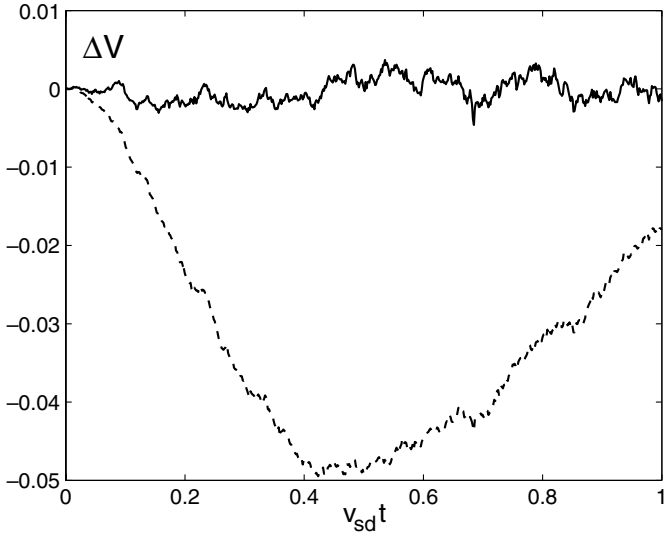

Fig. 6. Simulation results for the electron beam slowing-down process (13). Difference in results from the new and conventional codes (solid line), and from the new code and analytic calculations (dashed line). Here, $\Delta V = \frac{V_a - V_{new}}{V_{e,0}}$, a denotes analytic, or simulation results from the conventional code.

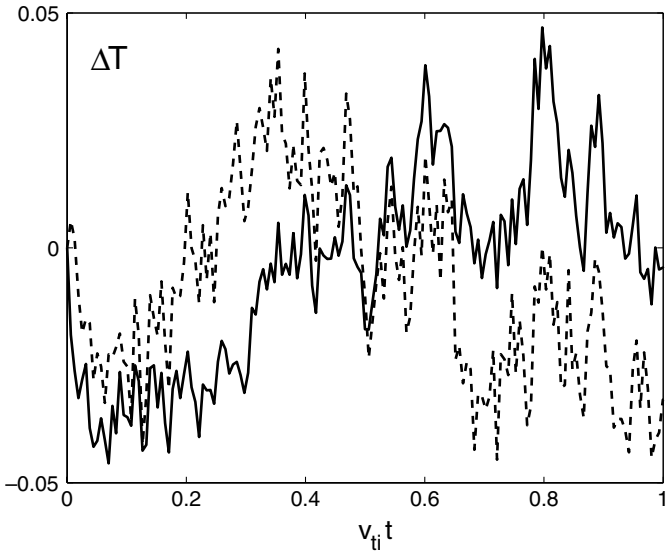

Fig. 7. Simulation results for the electron temperature isotropization process (14). Difference in results from the new and conventional codes (solid line), and from the new code and analytic calculations (dashed line). Here, $\Delta T = \frac{(T_\parallel - T_\perp)_a - (T_\parallel - T_\perp)_{new}}{(T_\parallel - T_\perp)_0}$.

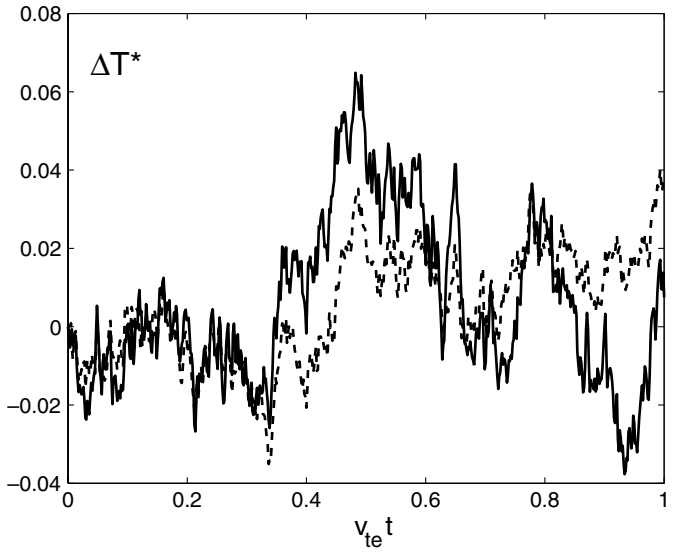

Fig. 8. Simulation results for the thermal equilibration process (17). Difference in results from the new and conventional codes (solid line), and from the new code and analytic calculations (dashed line). Here, $\Delta T^* = \frac{(T_2-T_1)_a-(T_2-T_1)_{new}}{(T_2-T_1)_0}$.

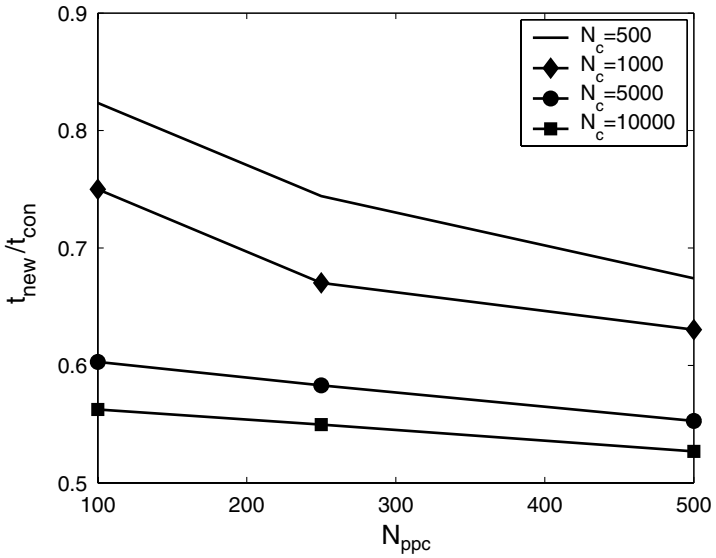

Fig. 9. Relative CPU time needed for simulation of thermal equilibration process (17) versus number of particles per cell, N_{ppc}, for different number of cells, N_c. The field solver is switched off and time steps are $\Delta t_c = \Delta t \ll (v_{te})^{-1}$.

The relative CPU time for the conventional and new operators is plotted in Fig. 9. As one can see, the CPU time decreases nearly linearly with increasing number of simulation particles and number of cells.

4. Conclusions

The above presented simulations demonstrate that the reduction of CPU time is similar for the PIC and MC (Coulomb collision) parts. Hence, the same reduction is expected for complete PIC–MC simulations, when both parts of the code are employed. In addition, we have redone a simulation of a collisional magnetized fusion plasma edge, presented in [24], with this optimized code, demonstrating a reduction of the CPU time of 45%. The simulation parameters were as follows: $N_{ppc} \sim 120$, $N_c = 6000$, $\Delta t_c = 300\Delta t$.

This analysis demonstrates that the conventional PIC codes can be optimized without any significant efforts. The optimized code keeps the accuracy of the original one and consumes up to 50% less CPU time.

Comparison with existing models of particle sorting [5,17,18] shows that for collisionless plasmas we get the same order of magnitude for speed-up. An exact quantitative comparison is complicated, as the gain in CPU strongly depends on the given processor and simulated plasma properties. The new approach presented here has the following advantages: (i) during the sorting not any operation is applied to the majority of particles (i.e. those particles which do not leave a given cell), (ii) the speed-up is valid for collisional plasmas too. Generalization for parallel computations is obvious, as particles are already "sorted" during whole simulation.

The introduced updated binary Coulomb collision operator can be used in any code, which will allow reducing significantly the number of calls of the random number generator and speed-up the simulation. We expect, that this model can be easily generalized for other nonlinear collision operators (e.g., for neutral particle collisions), requiring particle sorting according to grid cells. The detailed description of these operators will be the subject of our future work.

Acknowledgments

R. Schneider acknowledges funding of the work by the Initiative and Networking Fund of the Helmholtz Association. D. Tskhakaya acknowledges support through the SFB-TR24.

References

[1] J.P. Verboncoeur, Plasma Phys. Contr. Fusion 47 (2005) A231–A260.
[2] E. Kawamura, C.K. Birdsall, V. Vahedi, Plasma Sources Sci. Technol. 9 (3) (2000) 413–428.
[3] C.K. Birdsall, A.B. Langdon, in: S. Rao, M. Eichenberg (Eds.), Plasma Physics Via Computer Simulation, McGraw-Hill, New York, 1985.
[4] V.K. Decyk, Comput. Phys. Commun. 87 (1–2) (1995) 87–94.
[5] V.K. Decyk, S.R. Karmesin, A. de Boer, P.C. Liewer, Comput. Phys. 10 (1996) 290–298.
[6] R.W. Hockney, J.W. Eastwood, in: A. Hilger (Ed.), Eastwood Computer Simulation Using Particles, IOP, Bristol and New York, 1999.
[7] T. Takizuka, H. Abe, J. Comput. Phys. 25 (3) (1977) 205–219.
[8] C.K. Birdsall, IEEE Trans. Plasma Sci. 19 (2) (1991) 65–85.
[9] V. Vahedi, M. Surendra, Comput. Phys. Commun. 87 (1995) 179–198.
[10] W.X. Wang, M. Okamoto, N. Nakajima, S. Murakami, J. Comput. Phys. 128 (1) (1996) 209–222.
[11] K. Nanbu, Phys. Rev. E 55 (1997) 4642–4652.
[12] A.J. Christlieb, R. Krasny, J.P. Verboncoeur, Comput. Phys. Commun. 164 (2004) 306–310.
[13] D. Tskhakaya, S. Kuhn, Contrib. Plasma Phys. 42 (2–4) (2002) 302–308.
[14] J.P. Verboncoeur, M.V. Alves, V. Vahedi, C.K. Birdsall, J. Comput. Phys. 104 (2) (1993) 321–328.
[15] D.V. Anderson, D.E. Shumaker, Comput. Phys. Commun. 87 (1995) 16–34.
[16] T. MacFarland, H.M.P. Couchman, R.F. Pearce, J. Pichlmeier, New Astron. 3 (1998) 678–705.
[17] K.J. Bowers, J. Comput. Phys. 173 (2001) 393–411.
[18] R.J. Thacker, H.M.P. Couchman, Comput. Phys. Commun. 174 (2006) 540–554.
[19] O.V. Batishchev, X.Q. Xu, J.A. Byers, R.H. Cohen, S.I. Krasheninnikov, T.D. Rognlien, D.J. Sigmar, Phys. Plasmas 3 (9) (1996) 3386–3396.
[20] O.V. Batishchev, S.I. Krasheninnikov, P.J. Catto, et al., Phys. Plasmas 4 (5) (1997) 1672–1680.
[21] A. Bergmann, Contrib. Plasma Phys 38 (1998) 231.
[22] R. Shanny, J.M. Dawson, J.M. Greene, Phys. Fluids 10 (6) (1967) 1281–1287.
[23] D.L. Book, NRL Plasma Formulary, Naval Research Laboratory, Washington, DC, 1978.
[24] D. Tskhakaya, A. Loarte, R.A. Pitts, M. Wischmeier, S. Kuhn, in: J.W. Connor, O. Sauter, E. Sindoni (Eds.), Theory of Fusion Plasmas, Societa Italiana di Fisica Bologna, Italy, 2004, pp. 97–109.