ASTRONOMY & ASTROPHYSICS
SUPPLEMENT SERIES

FEBRUARY II **2000**, PAGE 157

Astron. Astrophys. Suppl. Ser. **142**, 157–163 (2000)

Parallelized tree–code for clusters of personal computers

H.R. Viturro and D.D. Carpintero

Facultad de Ciencias Astronómicas y Geofísicas, Universidad Nacional de La Plata, and Consejo Nacional de Investigaciones Científicas y Técnicas de la República Argentina, Argentina
e-mail: ddc@fcaglp.unlp.edu.ar

Received June 29; accepted November 12, 1999

Abstract. We present a tree–code for integrating the equations of the motion of collisionless systems, which has been fully parallelized and adapted to run in several PC–based processors simultaneously, using the well–known PVM message passing library software. SPH algorithms, not yet included, may be easily incorporated to the code. The code is written in ANSI C; it can be freely downloaded from a public ftp site. Simulations of collisions of galaxies are presented, with which the performance of the code is tested.

Key words: methods: numerical — galaxies: interactions

1. Introduction and motivations

In recent years, personal computers (PCs) became very efficient computational devices. Processors of the last generation PCs can rival nowadays with those of the more traditional workstations, with the additional advantage of being cheaper. As a consequence, clusters of PCs with their processors connected in parallel turn out to be a relatively economical way to reach a very high performance of computation. (In particular, this becomes the only accessible tool in those places with limited access to computational resources). Obviously, such a parallel arrangement is not suitable to the running of a classic sequential code; thus, from the appearance of clusters of PCs has arisen the necessity of developing strategies for programming in parallel.

Besides this, it is well known that discretization problems arise when simulating a large system such as a typical galaxy ($\simeq 10^{11}$ stars) with sets of 10^4 or even 10^5 particles, see for example Hernquist & Barnes (1990): whereas a typical star moves within a smooth potential in a real galaxy, a typical particle suffers multiple collisions during the simulation, thus resulting in spurious relaxation effects. Thus, a large number of particles is needed in order to properly simulate such systems. As the number of particles grows, so does the computational time involved, and fast machines and efficient algorithms become vital. Clusters of PCs and parallel algorithms come to satisfy these needs.

The paper is organized as follows: Sect. 2 describes the sequential features of the code, whereas Sect. 3 deals with those aspects concerning parallel programming. Some tests and simulations are presented in Sect. 4. The conclusions are considered in Sect. 5.

2. Description of the code

We have adopted the basic algorithm of the tree code described by Barnes & Hut (1986, 1989), Hernquist (1987), and Barnes (1995).

2.1. Computing accelerations

The first step in computing the acceleration of a particle in a tree code is the construction of the tree structure. Following the foregoing authors, we have adopted the octal–tree scheme. The construction itself begins identifying the largest cell (the *root*) of the tree with a cube containing the entire system of particles. This cell is subdivided into eight equal subcells; these subcells are in turn recursively subdivided into eight cells each, and so on until the ending cells (the *leaves*) contain either one or none particles.

Next, total masses, center of mass coordinates, critical radii, and quadrupole moments are calculated for each cell by recursively descending the tree. A fundamental aspect in this phase is the use of the so–called "parallel axis theorem" (Hernquist 1987), which allows the computation of quadrupole moments in a recursive way.

Send offprint requests to: D.D. Carpintero
Correspondence to: Observatorio Astronómico – Paseo del Bosque S/N, 1900 La Plata, Argentina.

The last step in computing the acceleration of a particle starts traversing the tree from root to leaves. A cell will contribute as a whole to this acceleration if it is far enough from the particle. Otherwise, the cell is discarded and the proccess is repeated with its children. The walk ends when all subdivided cells contain either one or zero particles. To decide whether or not a cell must be subdivided, a multipole acceptance criterion (MAC) is used (Salmon & Warren 1994). The original MAC (Barnes & Hut 1986) is based on the aperture angle θ, i.e. the ratio between the size of the cell and its distance from the particle being accelerated: a cell is subdivided whenever θ is greater than certain threshold. We have incorporated in our code the three MACs commonly in use: the original proposed by Barnes & Hut (1986), the $b_{\rm max}$ criterion (Salmon & Warren 1994), and the modified Barnes criterion (Barnes 1994).

2.2. Time integration

On most tree codes, the second–order symplectic leap–frog algorithm is the common choice for integrating the equations of motion. However, we have also tried other methods, namely symplectic integrators of fourth and sixth order e.g., Kinoshita et al. (1991), by integrating an N–body system modelling a King's sphere (King 1966; Binney & Tremaine 1987) with $N = 10000$ particles and central relative potential $W_0 = 5$. No better conservation of the energy nor angular momentum was achieved when using the latter integrators. Moreover, for each integration step, the acceleration had to be computed several more times than with the leap–frog. Thus, no advantage was observed in raising the order of the integrator. In addition, the leap–frog is very simple to code and does not require large amounts of memory, as other methods do.

3. Description of the parallelized code

A computational problem is traditionally solved step by step, by means of a unique processor; we say in this case that the problem is solved *sequentially.* In contrast, when a set of processors are connected (e.g., through a network) in order to solve a problem collectively, we talk about *distributed* or *parallel computation.* The development of a program which operates under distributed computation comprise three basic steps.

The first one consists in examining minutely the problem at hand, searching for and separating those (generally small) parts suitable for simultaneous processing. The specific way this is worked out is intimately related not only to the physical problem to be solved, but also to the kind of numerical algorithms to be used. In the case of an N–body problem, the splitting may be made quite simply. Each processor is given the data of the entire set of particles, so they can each generate the whole tree structure. However, each processor computes the accelerations and updates the positions and velocities of only a subset of particles. At the end of each time–step, the processors interchange information, receive the new data of all the particles, construct the new tree structure, and continue the integration of the fresh set of particles each one was assigned to work with. This way of parallelization may not seem optimal (in fact, it is not); however, the CPU time wasted generating a new whole tree is a small fraction of the CPU time invested in the computation of the accelerations. For example, for $N = 100\,000$, this fraction is approximately 1/120. It is worth to note that, if a leap–frog integration algorithm is used, the time needed for updating the positions and velocities is also negligible with respect to the computation of the accelerations.

Besides that, the environment in which our processors work does not conform an ideal situation, indeed: a few PC–ix86's, not dedicated exclusively to the integration, and connected by a 10 Mbits/s ethernet net shared with more than 100 other PCs. Therefore, the load over each machine shifts continuously. Unfortunately, there is no easy way to improve the distribution of tasks under such circumstances; refined algorithms trying to balance the load based solely on the dynamics of the problem will not benefit the overall performance. However, to get the best, our program does a dynamical balancing of the distribution of particles based on the actual load of each processor, trying to reach an even distribution of real times (as opposed to CPU times) on each one.

The second step in developing a parallel program is the choice of the programming methodology, which in turn determines the logical structure of the program. There are two basic models: the master–slave and the fork–join schemes (Geist et al. 1994). In the first one, a master program supervises the running of a group of slave programs, controlling also their interchange of information; the slaves do the actual computation. Thus, two essentially distinct programs must be maintained. On the other hand, the fork–join scheme makes use of a unique program, replicated on each processor. Depending on how it was first called, this program acts as a *parent* program on one of the processors, or as a *child* in the others. The children receive data from the parent and compute; the parent not only distributes tasks, but may also compute if desired. We have chosen the latter alternative, mainly because it allows to maintain and upgrade the software more efficiently than with the former method.

The third step is to select the tool with which the processors will comunicate themselves. Up to now, there are two paradigms: the Parallel Virtual Machine (PVM), and the Message Passing Interface (MPI) (Geist et al. 1996). Although the MPI is recognized as a future standard, at the time our program started to be assembled there were several versions of it at hand, differing appreciably with respect to installation procedures and operation. The PVM, on the other hand, had a unique version, permanently mantained and upgraded until now. Moreover, its operation is simple and flexible, allowing a more comfortable implementation and use. Thus the PVM was our choice.

3.1. Implementation details

Here we describe the logical structure of our program *poctgrav*, pointing out those aspects concerning the parallel features.

When running over one processor only, *poctgrav* works fine as a normal sequential program. However, the PVM sofware must be installed properly even in this case for the program to run, because the user might want to add other processors later.

When running over more than one processor, *poctgrav* is launched from one of them: the *mainhost*. This task, having contacted the PVM daemon, learns it is the parent. Once the parent is running, the PVM searches for the presence of any other processor requested by the parent, and launches on each of them a previously stored copy of *poctgrav*, i.e., the children. Should a processor fail to be contacted at this starting phase, the PVM sends a warning and everything is stopped. Next, each child communicates with its local PVM daemon to recognize whether it is the parent or a child (learning, of course, they are children). Thus, every task knows whether it has to give orders or it has to wait and execute orders. Once the integration has begun, the failure of a processor other than the parent (e.g., a machine breaks down), does not stop the program. Instead, *poctgrav* resumes execution from the last time step, taking into account which child was lost.

The following is a pseudo–code skeleton of the main loop of the program.

```
int main(int argc, char* argv[])
  {
  /*——Initialisation————*/

  /* Contact the local PVM (mytid is an
    internal variable) */
  mytid=pvm_mytid();

  /* Stop if the PVM is not running */
  if(mytid< 0)return -1;

  /* Learn whether it is the parent
    (mygid= 0) or a child (mygid> 0).
    The string GROUPNAME is defined by
    poctgrav, and it is the same for
    all the tasks involved. */
  mygid=pvm_joingroup(GROUPNAME);

  /* if this task is the parent */
  if(mygid==0){

    /* Read control parameters for the
      simulation. Also, instruct the
      task whether it is a parallel
      run and which processors will
      be its children. */
    read_params();

    /* Verify that a PVM daemon is alive
      in each host. Start any which
      is not. Then start children
      processes. */
    activate_hosts_and_task();

    /* Read initial conditions. */
    read_bodies();

    /* If in parallel mode, transmit
      data to the children. */
    if(childrenactive){
      /* control parameters */
      send_params();
      /* initial conditions */
      send_bodies();
    }
  }
  /* if this task is a child */
  else{
    /* receive control parameters */
    receive_params();
    /* receive initial conditions */
    receive_bodies();
  }
  /*——-End initialisation———*/
  /*——-Begin integration———*/
  /*—of the equations of motion—*/

  /* Create and go over the tree;
    compute potential and accel.
    only for particles assigned to
    this processor. */
  calculate_phi();

  /* If operating in parallel mode,
    interchange data */
  if(childrenactive){

    /* if parent... */
    if(mygid==0)

      /* ...collect information. */
      receive_bodies();

    /* If child... */
    else
      /* ...send information.
      send_bodies();
  }
  /* Print data */
  if(mygid==0) print_statistic();

  /* Do the actual integration */
  leapfrog();
}
```

The leapfrog routine deserves a little more inspection:

```
int leapfrog()
  {
  time=initialtime;
  /* Procceed until integration is
    complete */
  while(time<=finaltime){

    /* Update positions and velocities.
      If in parallel mode, do it
      only for the local particles. */
    new_vel(deltatime/2);
    new_pos(deltatime);

    /* Each processor needs to know
      the new coordinates of all the
      particles,in order to update
      its own tree */
    if(childrenactive){
      /* The parent... */
      if(mygid==0){
        /* ...receives lists from
          each child, */
        receive_bodies(); */
        /* and sends the complete list
          to every child. */
        send_bodies();
        /* Here is where bodies are
          distributed according to
          the actual load on each
          processor. */
      }
      /* Each child... */
      else{
        /* ...sends its list to the
        parent, */
        send_bodies();
        /* and receives the complete
        list. */
        receive_bodies();
      }
    }

    /* Rebuild the tree, and compute
      potential and acceleration for
      the local particles */
    calculate_phi();

    /* Synchronize positions and
      velocities. The parent does all
      the statistics, so interchange
      information for this purpose */
    new_vel(deltatime/2);
    if(childrenactive){
      if(mygid==0)
        receive_bodies();
      else
        send_bodies();
    }

    /* Print statistics and
      update time */
    if(mygid==0) print_statistic();
    time+=deltatime;
  }
}
```

4. Simulations

4.1. Performance of the code

It is a common practice to test a new N–body code by integrating the equations of motion of a simple system of particles known to be in a steady state, and studying such things as relaxation effects and conservation laws. Thus, we set up several N–body systems following King's phase–space distribution function. We set the central density parameter $W_0 = 5$ in all cases. The units were chosen so that the total mass $M = 1$, the dispersion parameter $\sigma = 0.762$, and the gravitational constant $G = 1$; with this set of units, the total energy of the system is $E = -1/2$, and the global dynamical time $t_{\mathrm{D}} = 1$, so we are able to compare our results with those of Hernquist & Barnes (1990) and Huang et al. (1993). Only three runs are commented here, namely those with $N = 4096$, $N = 15000$, and $N = 100000$.

All the runs lasted 10 dynamical times. In particular, we ran the $N = 10^5$ model over seven PCs, with the following processors: four Pentium/166 MHz, one Pentium Pro/200 MHz, one Pentium II/233 MHz and one Pentium II/266 MHz. To compute the CPU time demanded by the whole integration, we took the maximum among the CPU times of the processors in each cycle of integration, and summed up all these maxima to get the total. (If the program were perfectly balanced, all processors would finish simultaneously their respective tasks in each cycle; unfortunately, this could not be accomplished at all times, mainly due to the fact that our machines were not entirely dedicated to the integration. However, the performance was always near the ideal, since the program continually compensates the different third–party loads of

Table 1. CPU times used to integrate King's spheres, in hours

N	θ	Δt	PII 266	HP 735	HP 735 (H)
4626	1.0	0.0039	1.0	1.9	1.9
4626	0.1	0.0034	11.8	22.0	32.1
15238	1.0	0.0063	2.7	5.0	9.4

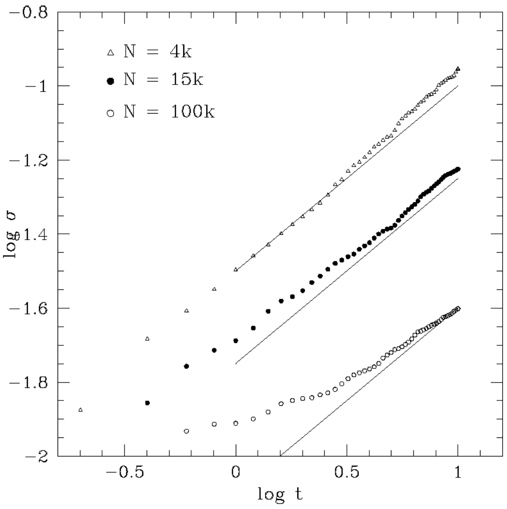

Fig. 1. Standard deviation of relative particle energies σ vs. time, for different number of particles. The straight lines have a slope of 0.5

the machines by redistributing adequately the number of particles assigned to each processor.) The integration of the ten dynamical times of the $N = 10^5$ run consumed 9.5 hours of CPU; it involved 1000 time steps (100 time steps per dynamical time).

Table 1 compares the speed of our code with that used by Hultman & Källander (1997). In all these experiments, only one processor was used. The fourth and fifth columns give the CPU times when using a Pentium II–266 MHz processor running a Linux operating system, and a HP 735 Workstation, respectively. The last column shows the times reported by Hultman & Källander (1997), whose code was also run on a HP 735 Workstation. The (fixed) time steps in our simulations (third column) were equal to the shortest individual time steps of the experiments of Hultman & Källander (1997). We can see that, despite this last disadvantage, the performance of our code increases with N, and with decreasing θ. This is probably due to a better run over the tree when computing accelerations.

In these preliminary simulations, all the standard tests were satisfactory (e.g., energy was conserved better than 3 10^{-4} in all cases). However, a test we found not to agree with previous results (Hernquist & Barnes 1990; Hultman & Källander 1997). This can be seen in Fig. 1, which shows the temporal behaviour of the standard deviation σ of relative particle energies

$$E_i = \frac{E_{\mathrm{f}i} - E_{0i}}{E_{0i}}, \tag{1}$$

where $E_{\mathrm{f}i}$ and E_{0i} are the final and initial energies of particle i, respectively. If the changes in energy were driven merely by a random walk diffusion process, the slope of $\log\sigma$ vs. $\log t$ would be 0.5 (Hernquist & Barnes 1990; Hultman & Källander 1997). However, as can be seen from the figure, the slope depends on the number of particles N, so relaxation is playing a role aside the diffusion due to the random accelerations. No dependence on the aperture angle θ or the time step Δt was found.

4.2. Colliding galaxies

Taking advantage of the speed of computation, we set up a pair of experiments in which two galaxies collide one another.

The first experiment was built with the aim of reproducing the Antennæ (NGC 4038/4039), a classical model to simulate since the pioneer work of Toomre & Toomre (1972). We therefore needed a model for spiral galaxies which remains stable at least during one dynamical time t_{D}, i.e., a period which suffices to obtain only those features caused by the collision, and not those caused by intrinsic evolution. To this end, we first followed Hernquist's (1993) recipe for building compound galaxies. However, this model proved not to be sufficiently stable to our experiment: when isolated, it evolves significantly well before the time at which the collision would begin. In most of our runs, a χ^2 test of this model yields $P(\chi^2) \simeq 1$ at only $t \simeq 0.2 t_{\mathrm{D}}$.

Therefore, we shifted to Barnes' (1992) model of compound galaxy. We used an exponential disc with $N_{\mathrm{d}} = 3072$ particles, mass $M_{\mathrm{d}} = 0.1875$, radial scale $R_{\mathrm{d}} = 0.083$, vertical scale $z_0 = 0.005$, and radial and vertical velocity dispersions in the ratio $\sigma_{\mathrm{R}}/\sigma_z = 2$. For the bulge, we set up a King's sphere with $N_{\mathrm{b}} = 1024$ particles, central potential $W_0 = 3$, total mass $M_{\mathrm{b}} = 0.0625$, and the scale of velocities $\sigma = 1$. Finally, for the halo, we used a similar King's sphere but with $N_{\mathrm{h}} = 16384$ particles, and total mass $M_{\mathrm{b}} = 4$. Thus, the halo, disc and bulge masses are in the relation 16:3:1, respectively, and their total mass is $M_{\mathrm{t}} = 4.5$. Fortunately, this compound galaxy proved to be stable at least during $1.75 t_{\mathrm{D}}$.

Once obtained a satisfactory model for the galaxy, we set up the initial conditions for the encounter leading to the Antennæ. We built a galaxy with 20480 particles as before, replicated it, and put both copies on the apocenter of a binary elliptical orbit with eccentricity $e = 0.5$ and pericentric distance $r_{\mathrm{p}} = 0.5$, with the standard Antennæ inclinations for their angular momenta e.g., Barnes (1988). We set the softening parameter $\varepsilon = 0.015$, the tree aperture angle $\theta = 0.7$, and a time step of $\Delta = 10^{-3}$. Figure 2 shows the initial conditions ($t = 0$) and the snapshot for which the experiment best resembles the sky–view of NGC 4038/39 ($t = 2.9 t_{\mathrm{D}}$). This simulation was run over six PCs (all the abovementioned machines, except one of those

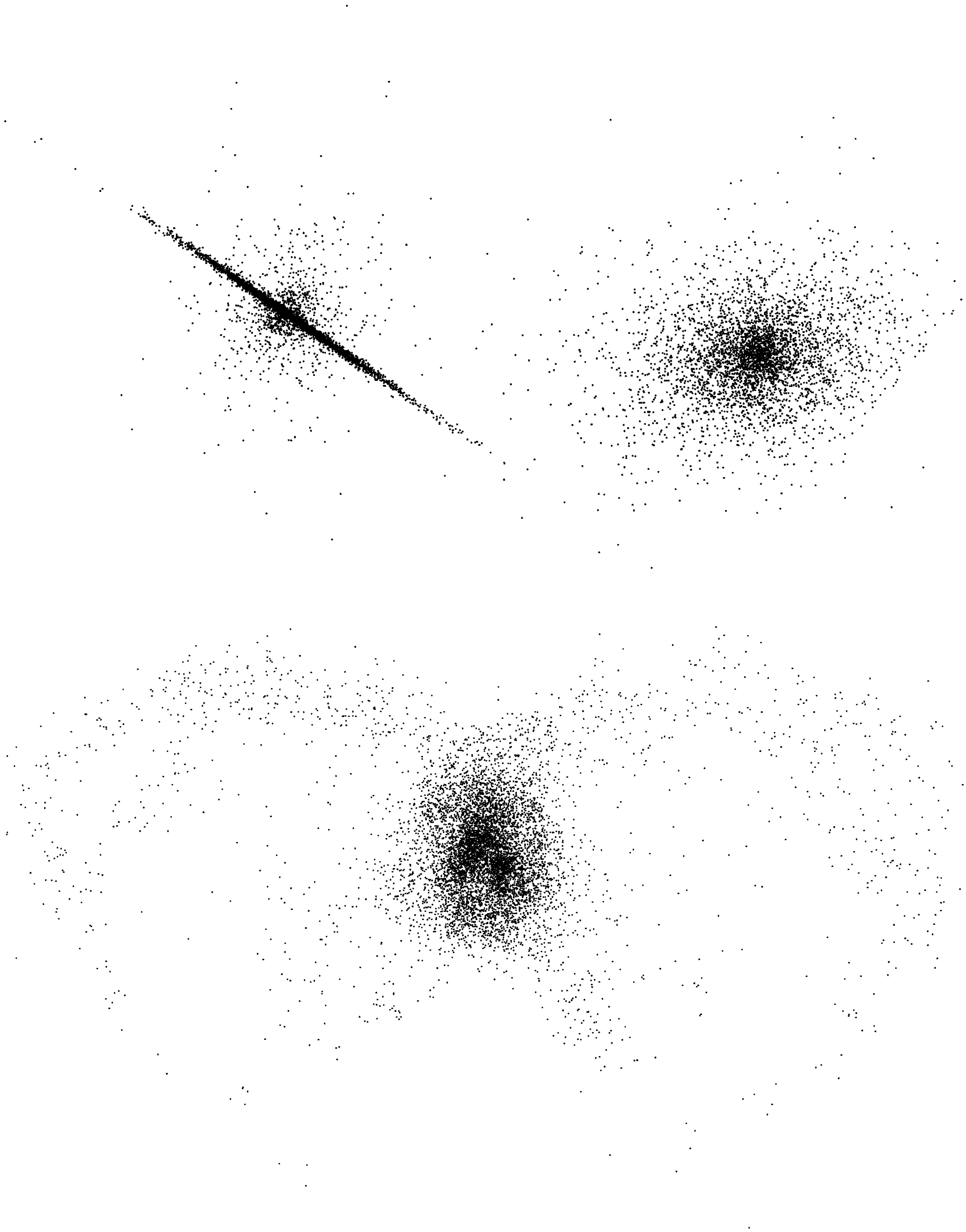

Fig. 2. Model of NGC 4038/39, the Antennæ. Top: initial configuration of the experiment. Bottom: intermediate ($t = 2.9$) state. The time and orientation were chosen in order to show the Antennæ as seen in the sky

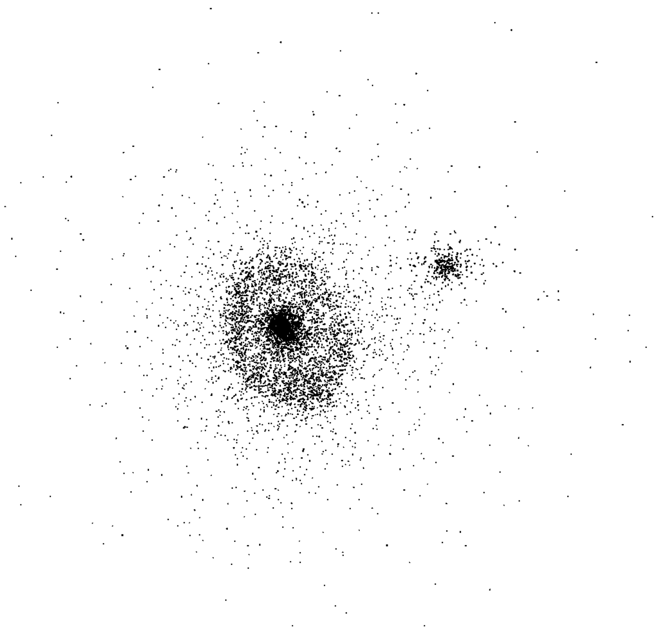

Fig. 3. Model of the Cartwheel galaxy

with the Pentium 166 MHz processor). The final time $t = 3.38t_{\rm D}$ was achieved after 5.47 hours of CPU time. In terms of *speed of the code* (Dubinsky 1996), i.e., the number of particles which could be evaluated per second, this simulation attained 2078 particles/s. (We disagree with the nomenclature here, because this is not really a measure of the speed of the code itself, but it depends on the number of processors involved. If we divide the *s*peed of the code by the number of processors, we get for our code 346 particles per second per processor; Dubinski's (1996) example, as a comparison, attains 375 particles per second per processor when using 16 processors.)

As a second simulation, we built up a King sphere with $W_0 = 12$, total mass $M = 2$, King radius $r_0 = 10^{-3}$, and $N = 512$, and threw it against one of the above (initial) compound galaxies, with $N = 6144$, in order to simulate the Cartwheel galaxy. The King's sphere was initially at 10 units away from the galaxy, i.e. at the outskirts of the halo. The relative velocity was $V = 2$ units, along the symmetry axis of the compound galaxy. Before choosing these parameters, a series of toy experiments with different masses, distances and velocities were run in order to achieve a good resemblance to the Cartwheel.

Figure 3 shows the final outcome of the simulation, from a point of view similar to that of the Earth. The experiment was run on one Pentium Pro/200 MHz and one Pentium II/266 MHz, and required 7.56 hours of CPU.

5. Conclusions

We have developed a PC–based parallel tree code, powerful and easy to use. It requires only the message passing library PVM installed, and a ANSI C compiler. The structure of the code was kept as simple as possible. Its power lies not only in its easy implementation and modularity —which allows the incorporation of, e.g., SPH–like hydrodynamics, but, more important, in that it was designed to work fine under hard conditions, i.e., on non–dedicated machines and through non–dedicated nets.

Thus, our program is able to determine the number of particles each processor should integrate based on its actual load. The differing loads may be due either to the features of the system being integrated, or, in the case of non–dedicated processors, to the sharing of CPU time with other processes. Thus, this optimization is of utmost importance for those places which cannot afford dedicated machines.

The program was tested by performing a number of standard experiments and simulations of collisions of galaxies, yielding satisfactory results in all cases. The complete ANSI C code, as well as an animation of the Antennæ experiment, is freely available via anonymous ftp to ftp.fcaglp.unlp.edu.ar in the subdirectory /pub/hviturro/poctgrav.

Acknowledgements. We would like to thank Drs. F.C. Wachlin and J.C. Muzzio for their useful advices.

References

Barnes J.E., 1988, ApJ 331, 699
Barnes J.E., 1992, ApJ 393, 484
Barnes J.E., 1994, in: Computational Astrophysics, Barnes J. et al. (eds.). Berlin: Springer–Verlag
Barnes J.E., 1995, in: The Formation of Galaxies, Muñoz-Tuñón C. & Sánchez F. (eds.). Cambridge: Cambridge University Press, p. 399
Barnes J.E., Hut P., 1986, Nat 324, 446
Barnes J.E., Hut P., 1989, ApJS 70, 389
Binney J., Tremaine S., 1987, Galactic Dynamics. Princeton: Princeton Univ. Press
Dubinski J., 1996, NewA 1, 133
Geist G.A., Beguelin A., Dongarra J., 1994, Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing. Cambridge: MIT Press
Geist G.A., Kohl J.A., Papadopoulos P.M., 1996, Calculateurs Paralleles, 8(2):-150, June (http://www.epm.ornl.gov/pvm/PVMvsMPI.ps)
Hernquist L., 1987, ApJS 64, 715
Hernquist L., 1993, ApJS 86, 389
Hernquist L., Barnes J.E., 1990, ApJ 349, 562
Huang S., Dubinski J., Carlberg R.G., 1993, ApJ 404, 73
Hultman J., Källander D., 1997, A&A 324, 534
King I.R., 1966, AJ 71, 64
Kinoshita H., Yoshida H., Nakai H., 1991, Cel. Mech. 50, 59
Salmon J.K., Warren M.S., 1994, J. Comp. Phys. 111, 136
Toomre A., Toomre J., 1972, ApJ 178, 623

List of objects

"NGC 4038/4039" on page 10
"NGC 4038/39" on page 12
"Cartwheel" on page 13
"Cartwheel" on page 14.