JOURNAL OF COMPUTATIONAL PHYSICS 96, 1–14 (1991)

A Dynamic Load-Balancing Algorithm for Molecular Dynamics Simulation on Multi-processor Systems

J. E. BOILLAT

Institute of Informatics and Applied Mathematics, University of Bern, Länggassstrasse 51, CH-3012 Bern, Switzerland

F. BRUGÈ

Department of Physics, University of Palermo, Via Archirafi 36, I-90123 Palermo, Italy

AND

P. G. KROPF

Institute of Informatics and Applied Mathematics, University of Bern, Länggassstrasse 51, CH-3012 Bern, Switzerland

Received November 22, 1989; revised June 5, 1990

A new algorithm for dynamic load-balancing on multi-processor systems and its application to the molecular dynamics simulation of the spinodal phase separation are presented. The load-balancer is distributed among the processors and embedded in the application itself. Tests performed on a transputer network show that the load-balancer behaves almost ideally in this application. The same approach can be easily extended to different multi-processor topologies or applications. © 1991 Academic Press, Inc.

1. INTRODUCTION

The balancing of the workload among the processors of a parallel computer system is central to achieve high performance. The load-balancing problem presents, in general, two aspects which have to be taken into account:

• The underlying architecture of the system used. This includes the topology of the multi-processor system and the possibilities of the processor interconnections. In the simplest case there is a predefined fixed and usually homogeneous topology, e.g., a hypercube scheme [1]. More flexible systems allow for a dynamic or static reconfiguration of the interprocessor connections [2]. Clearly the load-balancing problem is easier to solve if the topology is fixed and homogeneous during computation [3].

- The application to be run on the target system. The load-balacing problem is relatively easy to solve in the case of a homogeneous computational problem where each member of the underlying data domain is associated with a similar complexity in time (i.e., calculation time) and where the member distribution in the domain does not change during the computation (i.e., a static data distribution). This is the case, e.g., for spin lattice simulations [4] and equilibrium simulation of mobile objects [5, 6, 10]. For problems where either the data distribution or the member time complexity or both change during the computation, more complex methods have to be devised to *dynamically* reallocate the workload of the processors in order to achieve a well-balanced overall load of the system and thus a high efficiency [3].

In this paper we describe a new algorithm for dynamic load-balancing and its application to the molecular dynamics (MD) simulation of the spinodal phase separation of a two-dimensional fluid. The load-balancer is distributed among the processors and embedded in the application itself. The target system is a transputer network [7] organized as a ring; i.e., the topology is fixed and homogeneous. Thus the description and investigations of the algorithm will focus on the embedding and behavior of the algorithm in the chosen application of MD simulations. However, this approach is not limited to a particular multi-processor topology or application [8, 3]. Following the same approach a simpler algorithm has already been implemented which can be slower in critical conditions [9].

2. Molecular Dynamics Simulation on a Multi-processor System

The molecular dynamics simulation of the spinodal phase separation of a two-dimensional fluid involves systems where the range of correlations is many times the range of the potential, and very large numbers of particles (tens or hundreds of thousands) are required [6]. This kind of simulation can be handled on MIMD computers, by decomposing the simulation box into equal-area rectangular domains, and assigning one of them to each processor, thereby avoiding the need to consider widely-separated particle pairs [4, 11]. Such a domain structure for eight processors is shown in Fig. 1A, where a typical initial configuration of a system consisting of 4050 Lennard–Jones particles [12] is reported. The horizontal strips represent domains attributed to different processors. In what follows we shall indicate as *population* of a processor the number of particles belonging to its domain. In the exploded inset of Fig. 1, details of the boundary regions between two adjacent domains are shown. They are delimited by dashed lines one cutoff length apart from the domain boundary (solid line). Within these regions, particles belonging to a given processor interact with particles attributed to the neighboring processor. Therefore, the coordinates of these particles have to be exchanged between neighboring processors at each integration step. To make this communication efficient a ring topology is usually chosen for the inter-processor connections.

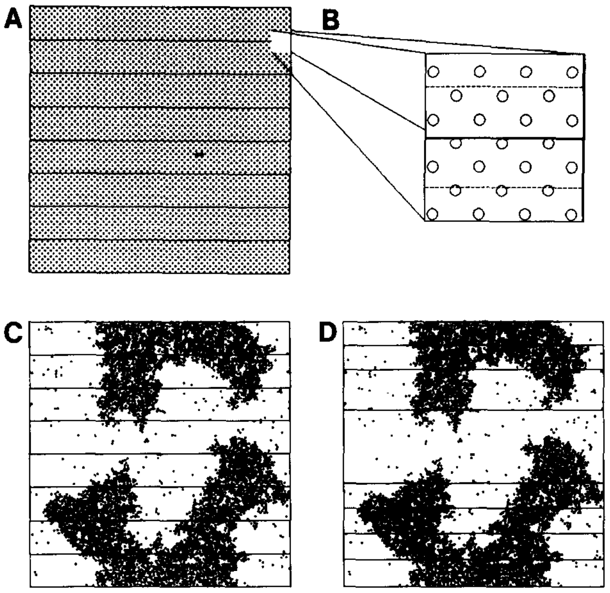

FIG. 1. (A) Typical initial configuration for the molecular dynamics simulation of a two-dimensional system involving 4050 Lennard–Jones particles, which undergoes a spinodal phase separation. The simulation box is subdivided into eight rectangular domains which are attributed to the eight ring-connected transputers constituting a MIMD computer. (B) Exploded view of the boundary region between two adjacent domains. (C) Final configuration reached after 100,000 integration steps (equivalent to a simulated time of 1 ns) keeping constant the domain limits. (D) The same final configuration as in (C) obtained while the load-balancer is active, which causes a different domain sizing in order to optimize the processor workload.

As the simulation proceeds, the system undergoes the spinodal separation developing a cluster structure, which could result in a performance impact, if the initial equal-area decomposition is maintained (Fig. 1C). Indeed, processors which happen to be responsible for domains containing few particles have only little to do; they update their particle coordinates and have to wait until the neighboring processors are ready to communicate. Therefore, to keep the overall performance of the parallel computer as high as possible throughout the whole simulation, we are faced with the problem of dynamically reallocating particles among processors.

3. Dynamic Load Balancing in an Application

To achieve the best possible overall performance of a parallel system during the entire duration of the application a load-balancer process is usually run concurrently with the main application [13]. This feature is generally assumed as essential for the design of a dynamic load-balancer [14], which then can be implemented according to many different schemes [13]. Among these, the method of stimulated annealing [15] can be used to search for the optimal load-balancing for "irregular" problems such as the simulation of phase transitions, assuming as the objective energy function to be minimized, the maximum workload per processor. However, since the annealing procedure steals computational time away from the principal computational task of interest, a difficult compromise has to be reached between the following requirements: the annealing procedure should be rapid enough to closely track the computation, which, on the other hand, should not be unreasonably slowed down by the procedure itself.

Common to these load-balancing methods is the *global* control of the processors' workload. The decision to rearrange the domains, i.e., to move particles from one processor to its neighbors in such a way that they will have about the same number of particles, is made by a central control process. This means that a dynamic load-balancer based on common optimization techniques, like simulated annealing, is driven by the minimization of a *global* function. This can strongly influence the efficiency of an application run on a distributed system, because the control process needs to know enough information about all the processors in order to make a balancing decision and it must advise the processor whether or not to exchange particles, i.e., to change their domain. These approaches thus imply a certain contradiction: the application is distributed over a multi-processor network without any global control, but the load-balancing requirements introduce a new globality again. In other words, the global control mechanism is contradictory to the ideas of parallel MIMD systems and algorithms.

These considerations prompted us to devise a new approach to load-balancing, where the load-balancer is distributed among the processors and where it is embedded within the application itself. According to this method, every processor along its normal computational cycle decides whether to give particles to its neighbors moving its upper and/or lower strip limits so as to maintain the population nearly equal to those of this neighbors. This decision is taken by each processor *locally*. As far as the load-balancer is concerned, each processor knows only the population of its immediate neighbors and its own. The domain of a processor is thereafter changed in order to reach a *local* equilibrium. In each computational cycle of the application *all* processors try in parallel to reach a local balance of the workload with respect to their neighbors. After a reasonable number of cycles, which in the worst case is *quadratic* to the number of processors, a global equilibrium, and thus a *globally* balanced situation, with populations of a similar size will be reached.

This approach has an analogy in physics: Imagine as a limiting case all particles assigned to one processor in the pipeline[1] and all the other processors having no population. Particles will dissipate by means of the load-balancer throughout the pipeline in a way that closely resembles the diffusion of heat in a bar without heat exchange with the environment. In fact it is possible to give a formal description of

[1] The distributed MD simulation algorithm is based on a ring topology to satisfy boundary conditions. However, for the load-balancing algorithm only a pipeline out of the ring is relevant (refer to Fig. 2).

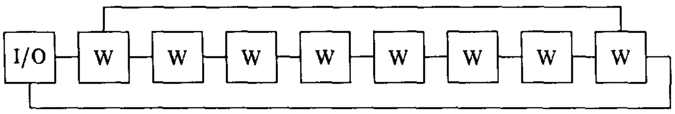

FIG. 2. Transputer network consisting of one I/O processor and eight worker processors. For the load-balancer only the pipeline of the workers is relevant, whereas for the MD simulation the workers' ring is used.

the proposed algorithm in terms of a *discrete Poisson equation* [8]. The numerical solution of a discrete Poisson equation indeed leads to the same iteration scheme as that for the proposed load-balancer.

The distributed load-balancing algorithm is designed such that, on each processor of the system, one and the same balancing procedure runs in parallel embedded in the application itself. Figure 3 shows the structure of the load-balancer algorithm in pseudo-code (Occam-like [16]). The diffusion of the load or the number of particles to be moved from one processor to another processor is controlled by the load differences between a processor and its neighbors and, additionally, by a weighing factor.

4. LOAD-BALANCED MOLECULAR DYNAMICS SIMULATION ALGORITHM

To understand how the dynamic load-balancer has been embedded in the MD algorithm it is worth noting that in any geometric parallel decomposition of local dynamics problems there is one main synchronization event among processors at

```
PROC load.balancer()
  SEQ
    ... initialize
    SEQ i = 0 FOR cycles
      SEQ
        ... application
        ... exchange load information with neighbors
        VAL old.population IS my.population :
        SEQ k = 0 FOR neighbors
          VAL to.move IS (old.population - neighbor.population[k])/weight :
          IF
            to.move > threshold
              SEQ
                my.population := my.population - to.move
                move.population[k] := to.move
            TRUE
              move.population := 0
        ... exchange move.population with neighbors
```

FIG. 3. Distributed load-balancer.

each integration step, i.e., the exchange of boundary and incoming–outgoing particles.

Each processor determines whether a particle has left its domain or is still remaining in it by checking its position with respect to the limits of its region just before exchanging particles. So, if the load-balancer wants to move particles from one processor to another it has to move the boundary limit between them *before* particles are recognized as internal or external to a processor domain and exchanged. The actuation part of the load balancer algorithm, i.e., the part in which the limits are actually moved must then precede the particle exchange procedure of the MD algorithm. On the other hand, if the load-balancer has to closely track the computation it has to work with up-to-date populations, i.e., with populations just computed *after* the particle exchange.

These considerations explain why the load-balancer code has been split into two parts whose relative position with respect to the main procedures of the iterative part of the concurrent MD program is shown in Fig. 4. Lines (folds) with comments in capital letter contain the actuation (`CHANGE BOUNDARY LIMITS`) and the decision part (`COMPUTE NEW BOUNDARY LIMITS`), respectively, of the load-balancer algorithm.

Acting in this way, the redistribution of particle coordinates among processors, required by the load-balancer, does not imply any further communication imposed synchronization event, varying solely the quantity of data normally transmitted in the particle exchange routine of the MD program.

To obtain the desired particle redistribution the load-balancer obviously cannot actually change particle positions but moves boundary limits between processors. Once the load-balancer has stated the amount of particles a processor has to send to its neighbor, it starts to diminish the processor's domain by trying to put backward the limit by a predefined small quantity (`delta` in Fig. 5) and checks at each trial how many particles would result outgone. As soon as the number of outgoing particles equals or exceeds the desired amount it stops and communicates the new limit to the involved neighbor. There is only one constraint the load-balancer has to take into account: the domains cannot become too small with respect to the boundary regions.

```
WHILE cycling
  SEQ
    ... calculate particle interactions
    ... update particle positions
    ... CHANGE BOUNDARY LIMITS
    ... exchange boundary and incoming - outgoing
        particles with neighboring processors
    ... adjust coordinate matrix to
        maintain the data structure
    ... COMPUTE NEW BOUNDARY LIMITS
```

FIG. 4. MD simulation with integrated load-balancer.

```
SEQ
  ...  exchange populations with neighbors

  to.move.down = (my.population - downstream.neighbor.population) / weight
  to.move.up   = (my.population - upperstream.neighbor.population) / weight

  --   compare with upperstream neighbor
  INT  outgoing.particles:
  BOOL try.again:
  IF
    to.move.up > threshold

      --  put backwards upper limit in steps of size 'delta'
      SEQ
        try.again := TRUE
        WHILE  try.again

          SEQ
            temporary.upper.limit := temporary.upper.limit - delta
            ...  count outgoing particles
            ...  test if the domain is becoming too small
            IF
              (outgoing.particles >= particles.to.give) OR domain.too.small
                try.again := FALSE
              TRUE
                SKIP

    TRUE
      SKIP

  ...  compare with downstream neighbor
```

FIG. 5. Load-balancer in the MD simulation.

5. Formal Description of the Load-Balancer

Formally, the multi-processor system used for the MD simulation, the *pipeline*, can be represented by an ordered set $T = \{1, 2, ..., n\}$ of processors. Let Q be the set of the particles to be distributed among the processors and let $l(i, t)$ be the particle population of the processor i at iteration step t. Since the MD simulation involves many particles, $l(i, t)$ may be considered as a real number in $[0, 1]$ (the total population will be normalized to 1, i.e., $\sum_{i=1}^{n} l(i, t) = 1 \ \forall t$). At each iteration step,[2] every processor will try to maintain its population nearly equal to those of its

[2] Because the load-balancer is embedded in the application, the iteration step t of the MD simulation and that of the load-balancing algorithm are identical.

neighbors. This can be achieved by exchanging $1/m$, $(m \geqslant 2)$ of the population difference with each immediate neighbor (m corresponds to the `weight` in the algorithm of Fig. 3), i.e.,

$$\frac{l(i,t)-l(j,t)}{m} \qquad (j=i+1 \text{ or } j=i-1).$$

This results in the following iteration scheme

$$\begin{aligned} l(i,t+1) &= l(i,t) - \frac{l(i,t)-l(i-1,t)}{m} - \frac{l(i,t)-l(i+1,t)}{m} \\ &= \frac{l(i-1,t)(m-2)\,l(i,t)+l(i+1),t)}{m} \qquad \text{if} \quad 1<i<n \\ l(1,t+1) &= l(1,t) - \frac{l(1,t)-l(2,t)}{m} \\ &= \frac{(m-1)\,l(1,t)+l(2,t)}{m} \\ l(n,t+1) &= l(n,t) - \frac{l(n,t)-l(n-1,t)}{m} \\ &= \frac{(m-1)\,l(n,t)+l(n-1,t)}{m}. \end{aligned}$$

The last two iteration equations are sufficient to keep the total amount of particles constant, i.e.,

$$\sum_{i=1}^{n} l(i,t) = \sum_{i=1}^{n} l(i,0) \qquad \forall t>0.$$

Let $L(t)$ be the vector $(l(1,t), l(2,t), ..., l(n,t))^{\mathrm{T}}$. Then each iteration step can be represented by the linear system

$$L(t+1) = PL(t) \qquad t \geqslant 0,$$

where

$$P = \frac{1}{m}\begin{pmatrix} m-1 & 1 & & & & & \\ 1 & m-2 & 1 & & & & \\ & 1 & m-2 & 1 & & & \\ & & \cdot & \cdot & \cdot & & \\ & & & 1 & m-2 & 1 & \\ & & & & 1 & m-2 & 1 \\ 0 & & & & & 1 & m-1 \end{pmatrix}.$$

THEOREM 1. *Let $L(0)$ be an initial distribution of particles, such that $\sum_{i=1}^{n} l(i,0)=1$, then if $m \geqslant 2$*

$$\lim_{t\to\infty} L(t)=\frac{1}{n}(1, 1, ..., 1)^{\mathrm{T}};$$

i.e., the iteration scheme converges towards the uniform distribution.

Proof. Note that $L(t)=P^{t}L(0)$. According to [17], the eigenvalues of P are

$$\lambda_k=\frac{(m-2)+2\cos((k-1)\pi/n)}{m}, \qquad k=1, 2, ..., n,$$

and the characteristic vector $v^{(k)}$ associated with λ_k has the following coordinates:

$$v_i^{(k)}=\cos(2i-1)\frac{(k-1)\pi}{2n}.$$

Let $u^{(k)}$ denote the normalized eigenvector $v^{(k)}/\|v^{(k)}\|$ associated with $v^{(k)}$. Since P is a symmetric matrix and all eigenvalues are different, the eigenvectors are orthogonal and it is easy to represent the initial particle distribution as a linear combination of the normalized eigenvectors:

$$L(0)=\sum_{k=1}^{n}\langle L(0), u^{(k)}\rangle u^{(k)}.$$

Let $\alpha_k=\langle L(0), u^{(k)}\rangle$. Since $u^{(1)}=(1/\sqrt{n})(1, 1, ..., 1)^{\mathrm{T}}$, $\alpha_1=1/\sqrt{n}$. Now note that $\lambda_1=1$ and that $|\lambda_k|<1$ if $k>1$ and $m\geqslant 2$. Since

$$P^{t}L(0)=\sum_{i=1}^{n}\alpha_i\lambda_i^{t}u^{(i)},$$

then

$$\lim_{t\to\infty} P^{t}L(0)=\alpha_1 u^{(1)};$$

i.e.,

$$\lim_{t\to\infty} P^{t}L(0)=\frac{1}{n}(1, 1, ..., 1)^{\mathrm{T}}. \quad \blacksquare$$

THEOREM 2. *The time complexity of the load-balancing algorithm is*

$$O(n^2).$$

Proof. The convergence speed of the iteration scheme depends in the worst case on the second greatest modulus λ of the eigenvalues of P, i.e., on

$$\frac{(m-2)+2\cos(\pi/n)}{m}.$$

Let $\varepsilon < 1$ be a suitable positive constant and let t be a positive integer, such that $\lambda^t \leqslant \varepsilon$. Note that for large n, i.e., for large networks,

$$\lambda = \frac{(m-2) + 2\cos(\pi/n)}{m} \simeq 1 - \frac{\pi^2}{mn^2}$$

Since

$$t = \ln \varepsilon / \ln \lambda,$$

it follows that

$$t = O\left(\frac{1}{\ln(1 - 1/n^2)}\right).$$

Since

$$\ln\left(1 - \frac{1}{n^2}\right) \simeq \frac{1}{n^2},$$

the complexity of the load-balancing algorithm is $O(n^2)$. ∎

6. Implementation and Results

The computer system on which the load-balancer has been implemented and tested is made up by eight 20-MHz T800 transputers (designated by W in Fig. 2) with 256 Kbyte RAM each, connected as a ring, plus a 20-MHz T800 transputer (designated by I/O in Fig. 2) with 2 Mbyte RAM which, being interfaced to the host IBM PC/AT, provides I/O capabilities to the transputer network.

The physical system chosen as test case for evaluating the performance of the load balancer consists of 4050 particles interacting by the Lennard–Jones 12–6 potential ($\sigma = 3.405 \mathring{A}$ and $\varepsilon = 0.23804$ Kcal/mole) which are enclosed in a square box of $381.36 \mathring{A}$ side length. The standard periodic boundary conditions are imposed and a potential cutoff of $9.56 \mathring{A}$ is assumed. A triangular lattice and a maxwellian velocity distribution corresponding to a temperature of 50.3 K are chosen as initial conditions. The temperature value is kept constant during the simulation by rescaling the particle velocities at each integration step.

The dynamics of the phase separation process has been followed to very late times (1 ns) and the effectiveness of our approach to load-balancing is evident from Fig. 6, where the execution time per MD step is reported with (continuous line) and without (+ symbols) the load balancer. As one can note the execution time per step increases somewhat even when the load-balancer is active. This is due to the increasing number of in-range interactions the processors have to evaluate as the particle condensation goes on. As Fig. 7 clearly shows the load-balancer has been

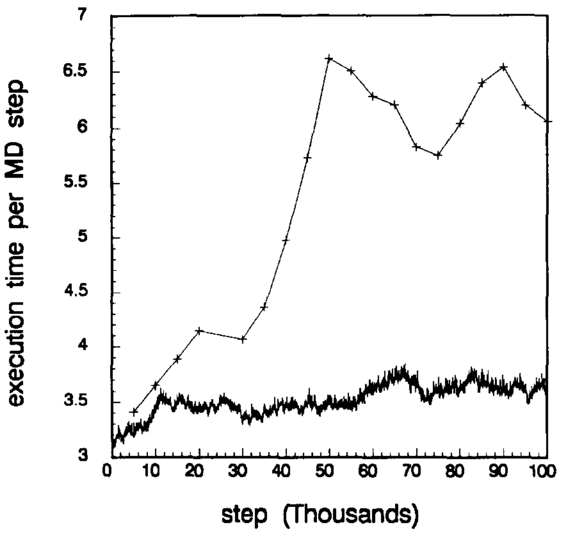

FIG. 6. Execution time per MD step vs. step number with (continuous line) and without (crosses) load-balancer.

able along the whole simulation to keep the maximum population over processors very near the ideal ratio of the particles to the number of processors (506 in our case). In this run a `weight` factor of 4 was selected and a weighted particle difference of 2 has been chosen as the triggering threshold for the load-balancer intervention (`threshold` in Fig. 3 and Fig. 5).

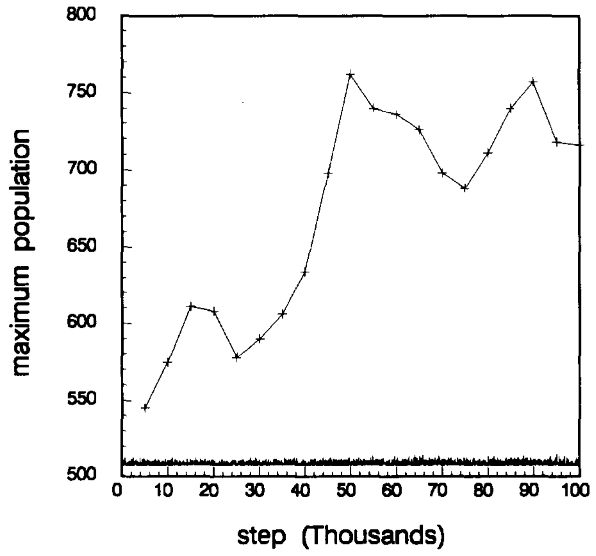

FIG. 7. Maximum population over processors vs. MD step number with (continuous line) and without (crosses) load-balancer.

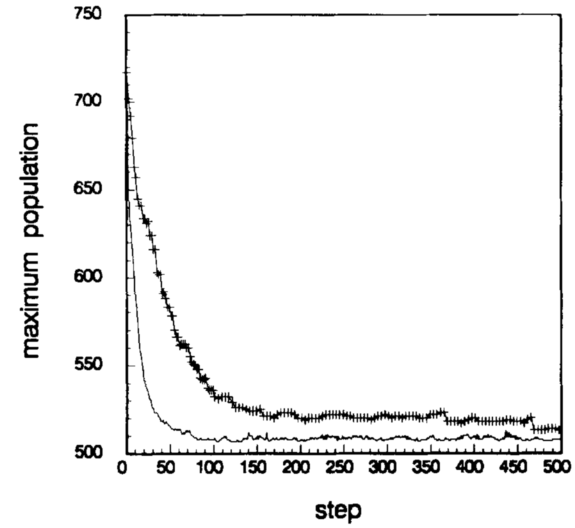

FIG. 8. Maximum population over processors vs. MD step number after the activation of the load-balancer (continuous line) starting from the domain distribution shown in Fig. 1C to reach the equilibrium population distribution of Fig. 1D. Crosses refer to the performance of a simpler load-balancing algorithm.

To give a show of the load-balancer abilities when applied to situations of large unbalance, two tests have been performed. In the first one, the load-balancer starts from the equal area decomposition shown in Fig. 1C and as one can see from Fig. 8 (continuous line) population equilibrium is attained within a few tens of steps. As a comparison, in the same figure the performance of a simpler load-balancing algorithm [9] is reported (crosses) when applied to the same initial situation. The second test starts from the same domain decomposition of the preceding one but

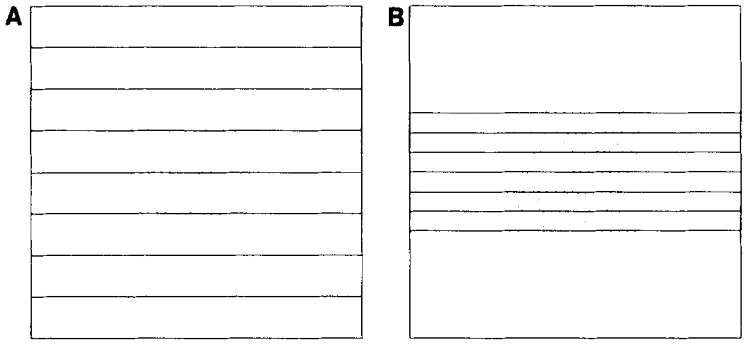

FIG. 9. Further example of the effectiveness of the presently described load-balancer which takes only 40 steps to reach the domain distribution shown on the right starting from the distribution depicted on the left. The number of particles placed at the center of the simulation box is 1024.

with 1024 particles concentrated at the center of the simulation box. A few tens of steps are needed by the load-balancer to pass from the unbalanced distribution of Fig. 9A to the balanced one shown in Fig. 9B.

7. Conclusion

In conclusion, the above described approach to the load-balancing problem of multi-processor systems features remarkable advantages as compared with more traditional methods. Indeed, the present load-balancer

- removes the need of making the difficult choice of how long and how frequently the load-balancer has to intervene, stealing away processors and computational cycles from the MD simulation
- automatically tracks closely to the computation, because it acts at each computational step
- does not require practically any additional communication imposed synchronization among processors, because the necessary information exchange for the load-balancer is completely embedded in the application and
- the parallel decomposition can be retained with this load-balancer also at the algorithmical level, since it acts locally with the same grain of parallelism as the MD simulation and does not need any global intervention.

A further important characteristic of this load-balancer is its rapidity in reaching equilibrium even when the initial configuration is very unbalanced. The implementation of the load-balancer has shown that the algorithm in most cases converges much faster than the worst case that complexity analysis has yielded. The presented load-balancing algorithm shows that a *global* characteristic, the load equilibrium, can be reached very efficiently by completely *local* information exchanges and calculations. Moreover, the proposed load-balancing technique can be applied to any application which can be decomposed in similar or equal parallel processes, and it can be adapted for an arbitrary processor network topology [3].

Acknowledgments

One of us (F.B.) acknowledges general support from Italian MPI—60%, and C.R.R.N.S.M fundings.

References

1. R. Hockney and C. Jesshope, *Parallel Computers* 2 (Hilger, Bristol, 1988).
2. J. Harp, C. Jesshope, T. Muntean, and C. Whitby-Strevens, "Phase 1 of the Development and Application of a Low Cost High Performance Multiprocessor Machine," in *ESPRIT* 86: *Results and Achievements*, Directorate General XIII ed. (Elsevier, Amsterdam, 1987).

3. J. Boillat and P. Kropf, "A Fast Distributed Mapping Algorithm," in CONPAR 90-VAPP IV, edited by H. Burkhart (Springer, Berlin, 1990), p. 405.
4. C. R. Askew, D. B. Carpenter, J. T. Chalker, A. J. G. Hey, M. Moore, D. A. Nicole, and D. J. Pritchard, *Parallel Comput.* **6**, 247 (1988).
5. D. Rapaport, *Phys. Rev. A* **36**, 3288 (1987).
6. F. Abraham, *Adv. Phys.* **35**, 1 (1986).
7. INMOS, *Transputer Reference Manual* (Prentice-Hall, Englewood Cliffs, NJ, 1988).
8. J. Boillat, *Concurrency: Practice and Experience* **2**, 4 (1990).
9. F. Brugè and S. Fornili, *Comput. Phys. Commun.* **60**, 39 (1990).
10. F. Brugè, V. Martorana, and S. Fornili, in *CONPAR* 88, edited by C. Jesshope and K. Reinartz (Cambridge Univ. Press, Cambridge, UK, 1989), p. 474.
11. F. Brugè and S. Fornili, *Comput. Phys. Commun.* **60**, 31 (1990).
12. B. Berne and G. Harp, *Adv. Chem. Phys.* **17**, 63 (1970).
13. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salomon, and D. Walker, *Solving Problems on Concurent Processors I* (Prentice-Hall, Englewood Cliffs, NJ, 1988).
14. G. Fox, A. Kolawa, and R. Williams, in *Hypercube Multiprocessors*, edited by M. Heath (SIAM, Philadelphia, 1987), p. 114.
15. S. Kirkpatrick, C. Gelatt, and M. Vecchi, *Science* **220**, 671 (1983).
16. INMOS, *OCCAM* 2 *Reference Manual* (Prentice–Hall, Englewood Cliffs, NJ, 1988).
17. M. Fiedler, *Linear Algebra Appl.* **5**, 299 (1972).