Journal of Computational Physics **161,** 250–263 (2000)
doi:10.1006/jcph.2000.6501, available online at http://www.idealibrary.com on IDEAL®

An Adaptive Load Balancing Method for Parallel Molecular Dynamics Simulations

Yuefan Deng,[*] Ronald F. Peierls,[†] and Carlos Rivera[‡]

Center for Scientific Computing, State University of New York, Stony Brook, New York 11794-3600; and [*]IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598
E-mail: [*]deng@ams.sunysb.edu, [†]peierls@ams.sunysb.edu, and [‡]criverac@ams.sunysb.edu

Received July 13, 1999; revised March 10, 2000

We describe an adaptive method for achieving load balance in parallel computations simulating phenomena which are distributed over a spatially extended region, but are local in nature. We have tested the method on standard short-ranged parallel molecular dynamics calculations. The performance gain we observe confirms the value of the method for this type of calculation. We discuss possible generalizations of the method, for example, to higher dimensions. © 2000 Academic Press

Key Words: load balance; parallel computing; molecular dynamics.

1. INTRODUCTION

We extend earlier work [1–3] on load balancing for parallel computations in modeling localized phenomena spreading over an extended region of two- or three-dimensional space. Modeling classical molecular dynamics with short-ranged interactions and solving partial differential equations are typical of such problems. The usual approach to parallelizing these computations is to decompose the spatial region into subdomains, one associated with each processor. The local nature of the computation helps ensure that communication costs remain small. We shall focus in this paper on the molecular dynamics problem. The computational work for a given processor at each time step depends on the number of particles in the corresponding spatial domain. This dependence will be somewhere between linear and quadratic, determined by the interaction range and the spatial distribution of the particles. (For partial differential equations, the corresponding parameters would be the number of mesh points and the choice of finite difference algorithm.)

For such computations it is natural to decompose the space by constructing a uniform rectangular grid, the spatial domains being the resulting rectangles (in 2D) or hexahedra (in 3D). In general, because the distribution of particles is not uniform, the computational loads will differ between domains, and the calculation will be inefficient as processors wait for the one with the heaviest load.

Our approach is to adaptively repartition the space by moving the vertices of the grid, maintaining its original topology but abandoning its uniformity and rectangular nature. We introduce an algorithm for determining how to move a given vertex depending on the relative loads of the domains which have that vertex in common. This vertex movement, as well as any resultant transfer of particle information from one processor to another, can be carried out in parallel. In general, because of the heuristic nature of the algorithm, complete balance is not achieved in a single step and the procedure must be iterated. Furthermore, as the simulation proceeds, the particles move and so the whole balancing procedure must be repeated at intervals.

The main contribution in this paper is the refinement of the algorithm presented in [3], its extension to 3D, and some test examples to investigate its performance. We have carried out three sets of tests. The first uses stationary synthetic data in 2D to explore and validate the method. The second simulates the motion of a set of noninteracting particles in 3D in order to explore issues of how many iterations are needed for a single balancing step and how often to rebalance. Finally the third involves a molecular dynamics simulation computing the trajectories of particles interacting under short-ranged Lennard–Jones forces. This demonstrates the degree of speedup achievable by our technique.

The idea of adaptively repartitioning by moving vertices has much in common with certain approaches to adaptive grid generation [1]. In particular, the algorithm described in Subsection 2.1 is similar to the mean value relaxation approach discussed in [4]. However, there are a number of crucial differences from the load balancing problem. For adaptive gridding, the objective is to match the density of grid points to the distribution of a well-defined weight function so as to achieve an optimal global error. For load balancing, where the weight function is the workload per processor, the objective is locally to minimize the workload of the busiest processor. If perfect balance is not achievable, it is irrelevant how the residual imbalance is distributed among processors. In addition, the workload is not a well-behaved function of position, so the careful analytical considerations which can be used for adaptive gridding are not available.

Other approaches to short-ranged molecular dynamics have also involved problem-dependent tiling of the spatial domain, but their methods of rearranging the grid are quite different from ours, and generally are restricted to two dimensions. One approach which has had some success is recursive bisection, but it is hard to carry it out efficiently with a constraint on the total number of subdivisions, and as the distribution evolves, such a decomposition does not change smoothly, but may need to be restarted from the top level.

In Section 2, we present the algorithm in 2D and 3D and indicate how it might be extended to higher dimensions.

In Section 3, we discuss the test cases: stationary tests in 2D and noninteracting and then full molecular dynamics in 3D, with particular attention to the latter.

In Section 4, we present the results of the tests.

2. ALGORITHM

As discussed in the previous section we focus on molecular dynamics. The region being simulated is partitioned by a logically rectangular grid into cells, each associated with a particular processor. For each simulated time step, the computational workload for a processor depends on the number and distribution of particles in the corresponding cell and its neighbors. We wish to move the grid points so that cells corresponding to the more

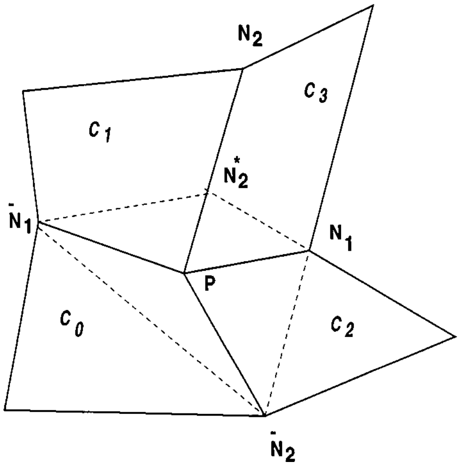


FIG. 1. Basic notation.

heavily loaded processors are reduced in size, in order to distribute the load more equally. To do this, we adopt a physical analogy imagining that each grid point is acted upon by "forces" directed toward the interior of more heavily loaded regions. The details of the algorithm have to do with how these fictitious "forces" are specified in terms of the workloads, and how the grid point is moved once the resultant "force" is known. Stability considerations require that the motion be constrained so as to maintain convexity of all the cells which result.

2.1. *Two Dimensions*

To illustrate the method, we begin with a 2D rectangular decomposition. Figure 1 illustrates the configuration of the grid at one particular point at some intermediate stage in the calculation. Let P be an arbitarary grid point, shared by four quadrilateral cells, $C_0, \ldots, C_3$. The neighboring grid points are $\bar{N}_1, N_1, \bar{N}_2, N_2$, as shown. Let W_i be the workload associated with cell i, and $W_{\max}$, $\bar{W}$ be the maximum and average values of W_i. Then $I_P = 1 - (W_{\max}/\bar{W})$ is a measure [3] of the degree of imbalance of these four cells.

One approach is to consider each of the four cells as a separate region and construct a "force" component into the interior of each cell for which $W_i > \bar{W}$. This is, in fact, the approach we adopt in the 3D case below. For 2D, however, we can get better stability if we consider the "force" components associated with pairs of adjacent cells. There are four of these, (C_0, C_1), (C_2, C_3), (C_0, C_2), and (C_1, C_3), but only two "force" components need be considered.

Consider first the two pairs (C_0, C_1), and (C_2, C_3). Only one of the two can have a total workload above average. Define

$$\delta_1 = \frac{(W_2 + W_3) - (W_0 + W_1)}{8\bar{W}},$$

where the sign of δ_1 determines which pair has the higher load. Let $\mathbf{v}_1$ be the vector from P to N_1, and $\mathbf{v}_{-1}$ be the vector from P to $\bar{N}_1$. Then we can define one component as

$$\mathbf{f}_1 = \begin{cases} |\delta_1|\mathbf{v}_1, & \delta_1 > 0 \\ |\delta_1|\mathbf{v}_{-1}, & \delta_1 < 0. \end{cases}$$

The second component $\mathbf{f}_2$ is constructed by comparing cells (C_0, C_2) with (C_1, C_3), and defining $\delta_2, \mathbf{v}_2, \mathbf{v}_{-2}$, and $\mathbf{f}_2$ in a similar way.

The shift in the position of vertex P is then given by the vector

$$\Delta_P = (1-\epsilon)\frac{\mathbf{f}_1 + \mathbf{f}_2}{2},$$

where ϵ is a parameter which controls how aggressively we seek to relax the grid. For loads which are fairly smoothly distributed within the cells, ϵ can be chosen to be quite small, but if it is possible that the majority of the excess load lies near the region boundary, then a larger value is called for.

We need to make several remarks:

(1) The above discussion assumes that P is an interior point. For boundary points we adopt reflective boundary conditions to extend the region, and apply the algorithm as before.

(2) If $W_0 \approx W_3 \gg W_1 \approx W_2$, then the imbalance I_P can be large, but both δ_1 and δ_2 small. This checkerboard case can be easily identified and then we introduce a third component $\mathbf{f}_3$, based on comparing (C_0, C_3) with (C_1, C_2). The vector directions are now

$$\mathbf{v}_3 = (\mathbf{v}_1 + \mathbf{v}_2)/2$$
$$\bar{\mathbf{v}}_3 = (\mathbf{v}_1 - \mathbf{v}_2)/2.$$

(3) It is essential that the new quadrilaterals which result from moving P remain convex. This will not necessarily be true; for example, in the configuration of Fig. 1, if the load in cell C_3 is sufficiently large, P can be moved so far into C_3 that the new cell C_2 will be concave. In order to prevent this, we replace N_2 by $N_2^\star$, the point where the extrapolated edge of C_2 intersects the line PN_2.

(4) In moving P we assume that all the neighbors N_k are held fixed. Other mesh points may be simultaneously moved without affecting the relocation of P. It is easy to see that we can apply a red-black coloring scheme to adjacent points and that in a parallel implementation half the total number of mesh points can be moved at each step.

(5) The approach can be carried out hierarchically. This involves an initial uniform decomposition into the coarsest mesh of exactly four cells. The interior point and boundary points are moved according to the algorithm to balance these loads. Each distorted rectangular cell which results is then uniformly subdivided into four subcells and the procedure repeated at the next level. This is a more efficient procedure if there exists significant imbalance at the coarsest level.

The main steps of the algorithm are:

Step I. For vertex P compute the shift in position according to the above prescription. Carry this out simultaneously for the set of grid points reachable from P by a succession of moves of two grid units in either direction.

Step II. Redistribute the load between processors according to this new decomposition. If P' is the new position of P, then the points P, P' together with each one of the neighbor points N form a triangle. Only the portion of the load lying inside these four triangles need be relocated. To test whether a given point lies inside a given triangle the results of three affine transformations of the point coordinates must all be nonnegative. The parameters of these transformations need only be computed once for each triangle. Verifying that a point lies inside a bounding rectangle containing the triangle is even simpler, so that the work of relocation is at worst linear in the number of particles in the cell.

Step III. Repeat these steps for the grid points held fixed in Step I.

Step IV. Repeat the above procedure until the desired balance (see Section 3) has been attained, or some specified cost has been exceeded.

2.2. *Three Dimensions and Generalization to Higher Dimensions*

The 3D version of the algorithm is fundamentally the same as the 2D version. However, there is an implicit asumption in the 2D case that is no longer true for higher dimensions. The four vertices of a quadrilateral in 2D uniquely determine its interior because there is a unique straight line between the two vertices specifying an edge. In 3D a hexahedron is not uniquely determined by its vertices. In general there is no single plane which passes through the four vertices specifying a face. Any three points do define a plane, and so the usual procedure is to construct the face as two triangles, with a dihedral angle between their planes. However, this is still not unique, since either of the two diagonals can be used to specify the triangular decomposition. For a logically rectangular grid, the choice of how to specify a face must be the same for both hexahedral cells sharing that face, but each such choice is independent. One way of resolving the ambiguity is to consider the subdivision of each hexahedron into five tetrahedra, and consider the grid to be a tetrahedral tiling of the region. In this case once the diagonal specifying one face of one cell has been chosen, the rest are uniquely determined, so that for any logically rectangular grid there are exactly two ways of representing it as a tetrahedral tiling.

We should note, however, that for the molecular dynamics problem we consider in this paper, it is not necessary to confront this ambiguity. We always begin with a strictly rectangular mesh for which the faces are in fact planes (the two triangular decompositions are identical, having zero dihedral angle). There is therefore no difficulty in making the initial assignments of particles to the interior of grid cells. At any subsequent stage, when the grid is no longer rectangular, our task is to determine which particles, initially in one particular cell, must be moved to another as a result of the movement of a vertex. In the 2D case, each edge, as the vertex moved, swept out a triangle, and it was necessary to determine which particles lay within those four triangles. In 3D, each face, as the vertex moves, sweeps out a tetrahedron, and all that is needed is to determine which particles lie within these 12 terahedra. This task is well defined and straightforward.

For the 2D algorithm there are four cells having any given grid point in common. There are also four pairs of cells having one of the edges through the grid point in common, and we focused on these pairs as the regions for balancing. In 3D there are eight cells sharing a given grid point, but only six groups of cells sharing an edge, so we would lose some discrimination power if we adopted the analogous algorithm. It should also be emphasized that the checkerboard type of imbalance which was most easily resolved in the context of considering pairs of cells is less likely to be a problem in 3D, since there are many more components which would have to cancel. More importantly, in the 2D examples, we studied static problems. For dynamic problems, such metastable configurations would only be transient and therefore less of a problem. For 3D, therefore, we focus on the individual grid cells as regions to generate “forces.”

The notation is a natural extension of the 2D case: P represents a general grid point, N_i, $\overline{N}_i$, $i = 1, \ldots, 3$, represent the neighboring points in each of the three grid directions, and $C_l, l = 0, \ldots, 7$, are the eight cells sharing the grid point P. The vectors from P to N_i, $\bar{N}_i$ are $\mathbf{v}_i, \mathbf{v}_{-i}$, respectively. If $(l_1 l_2 l_3)$ is the binary representation of l, then the cell C_l is spanned by the vectors $\mathbf{v}_{2l_1-1}, \mathbf{v}_{2l_2-1}, V_{2l_3-1}$. Thus, C_7 contains $PN_1N_2N_3$, C_5 contains $PN_1\bar{N}_2N_3$, etc.

The algorithm is then as follows:

Step I. For vertex P compute for each cell C_l the vector f_l by

$$\mathbf{f}_l = \begin{cases} \frac{(W_l - \bar{W})\sum_{i=0}^{3} \mathbf{v}_{2l_i-1}}{16W} & \text{if } W_i > \bar{W}, \\ 0 & \text{otherwise,} \end{cases}$$

where $l = 0, \ldots, 7$.

Step II. Construct the vector

$$\Delta_{\mathbf{P}} = (1 - \epsilon) \sum_{l=0}^{7} \mathbf{f}_l.$$

Step III. Move vertex P by the vector Δ_P.

Step IV. Proceed in the same manner with other vertices.

Step V. Reassign the load acording to this new decomposition.

Step VI. If the desired degree of balance (see Section 3) has not yet been attained, repeat the above steps unless improvement has ceased, or some specified cost has been exceeded.

3. TEST CASES: 2D STATIC SIMULATIONS AND 3D MD

We first studied the algorithm for simulations of static loads in 2D. The purpose was to see if the algorithm was able to find near optimum redistribution of loads under a variety of highly imbalanced initial cases. We were able to solve difficult and artificial problems such as the checkerboard distribution. More realistic tests were performed in 3D. Static tests were done to evaluate the capabilities of the algorithm and to set balance parameters for further tests. Next tests on molecular dynamics were performed to observe if the algorithm was also capable of working adaptively under changing load distributions. The ultimate goal was to observe whether the application of our load balance algorithm reduced the computational time required for MD problems. We report the load imbalance ratio and timing results. The load imbalance ratio is a measure how far the system is from perfect balance. It is defined in [3] as $I = 1 - \bar{W}/W_{\max}$, where $\bar{W}$ is the average load and $W_{\max}$ is the maximum load among all subdomains.

We also give some guidance on choosing the parameters for a particular problem. We remark that the parallel MD code used is a standard one. Each processor does an average of $N(N+1)/P$ force computations, where N is the total number of particles and P is the total number of processors. It does not implement a method such as linked cell that further reduces the force calculation count. The elements that are redistributed between processors are particles, while in a more efficient linked cell calculation one would redistribute linked cells or virtual domains [6].

4. TEST RESULTS

4.1. *Static 2D Simulations*

We show results for three artificial cases, each case involving 5×5 subdomains containing 1024 particles. The loads in the first two cases are fairly easy to balance and our algorithm finds a balanced decomposition after approximately 100 steps (see Figs. 2 and 3). The last

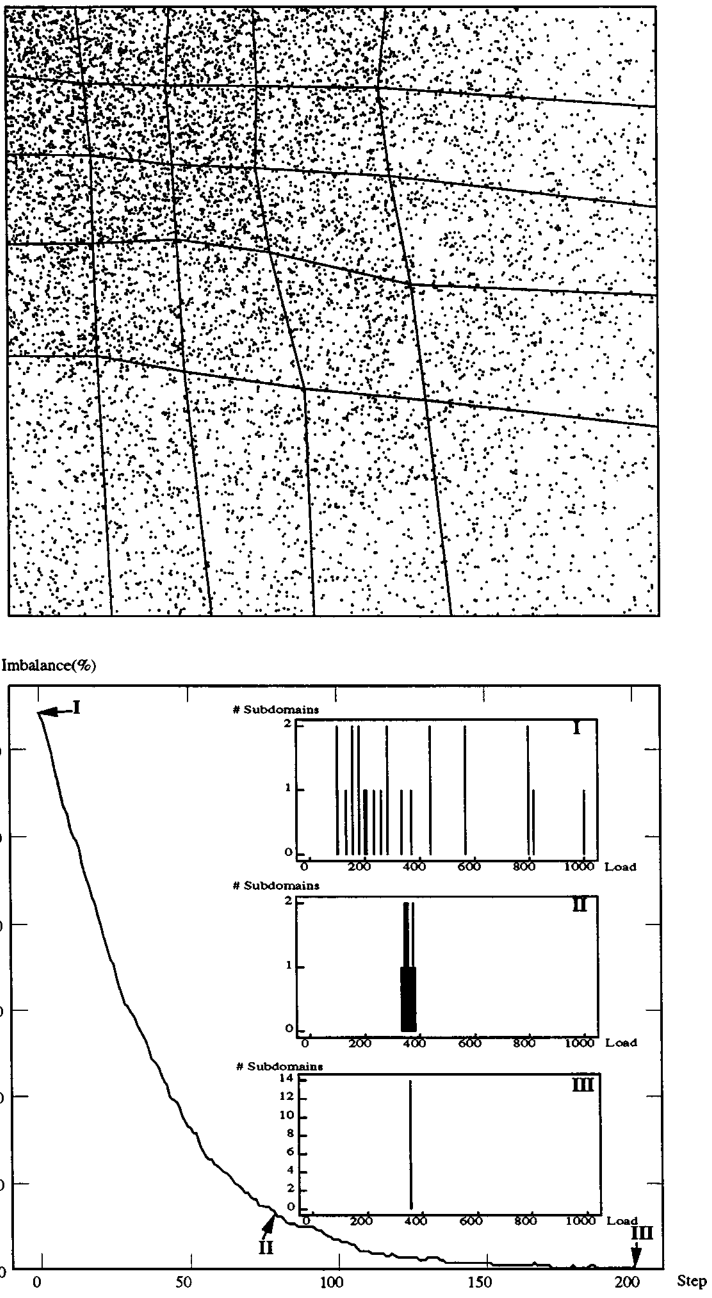


FIG. 2. (Top) This figure shows how we balance the loads on 5×5 subdomains with the upper left corner having the most load using uniform decomposition. (Bottom) Plot of the load imbalance ratio which falls from about 65% to nearly zero. Inset in this figure are three histograms showing the initial, midpoint, and final load distributions.

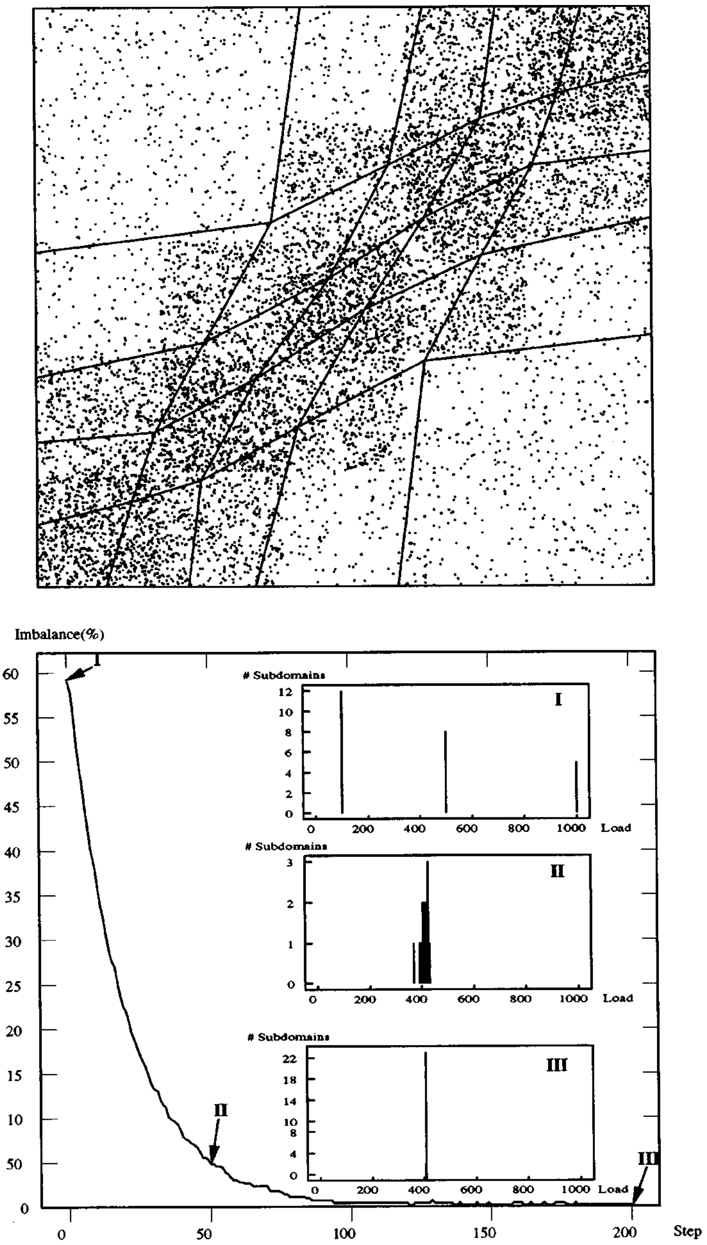
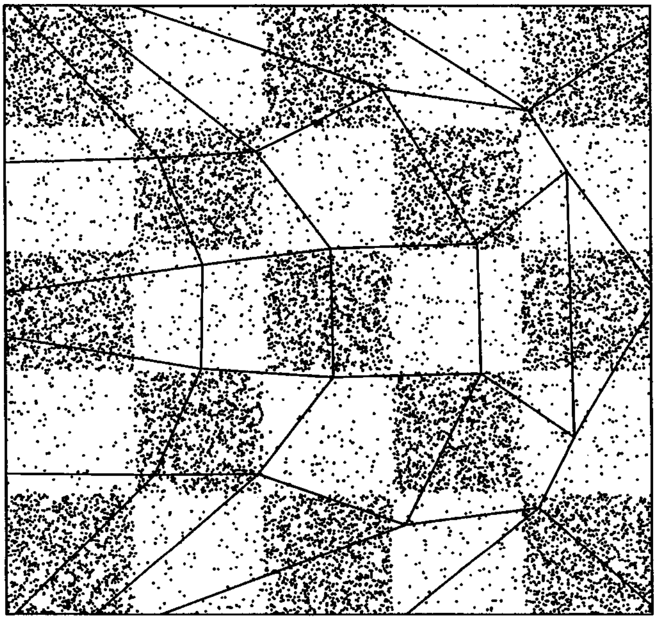


FIG. 3. (Top) This figure shows the solution for a dominant load along one of the diagonals. (Bottom) Plot of the load imbalance ratio which falls from about 60% to nearly zero. Inset in this picture are three histograms showing the initial, midpoint, and final load distributions.

case, involving checkerboard-like load distribution, is much more subtle to balance; the convergence to a balanced distribution is slower due to oscillation (see Fig. 4).

4.2. *3D MD Computations*

Problem I. The domain is decomposed to $3 \times 3 \times 3$ subdomains, containing 2048 particles, with reflective boundary conditions. Force is calculated but not used; particles move



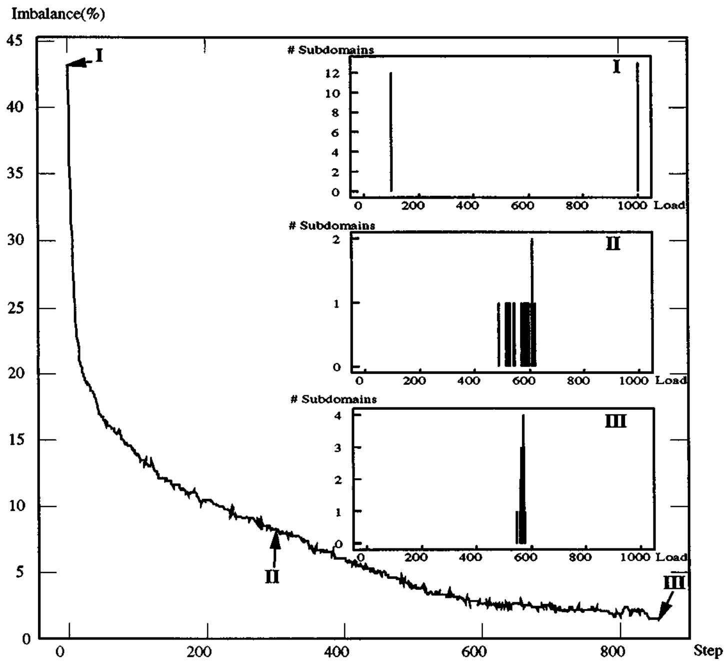


FIG. 4. (Top) This figure shows the solution for an initial load distribution in a checkerboard pattern. (Bottom) Plot of the load imbalance ratio which falls from about 44% to nearly zero. Inset in this figure are three histograms showing the initial, midpoint, and final load distribution.

by inertia (one can think of this as molecular dynamics for which the only acting force is gravity, and hence it is negligible). This helps create a greater load imbalance since the system is never in equilibrium. The result is an oscillatory load imbalance ratio for the dynamic case.

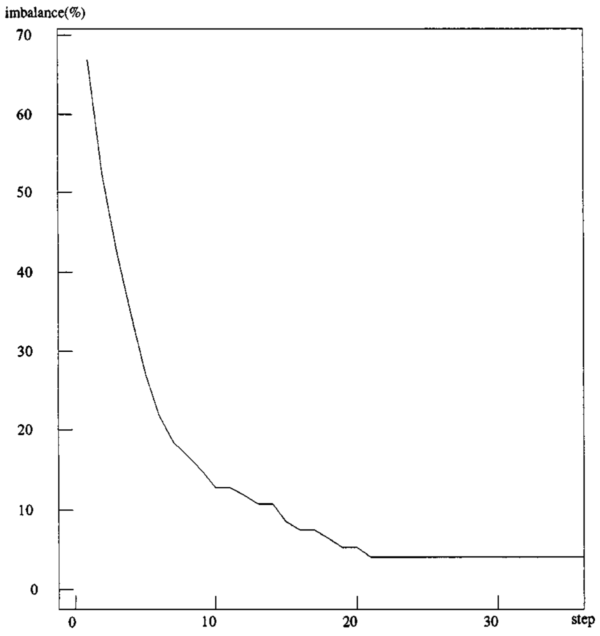


FIG. 5. Test 1 for Problem I. Load imbalance ratio vs balance iteration step. Static case, 35 balance iteration steps.

A static problem was performed several times until satisfactory balance parameters were found. The parameters were set as follows: $\epsilon = 0.1$, $S_{\max} = 20$, $I_g^{\text{TOL}} = 5\%$, $I_l^{\text{TOL}} = 10\%$. The parameter ϵ was introduced in Subsection 2.2 and it is used to control stability. While a large value for ϵ will slow the balance convergence an ϵ too small can create stability problems, in particular for rapidly varying loads. In our dynamic tests we found no stability

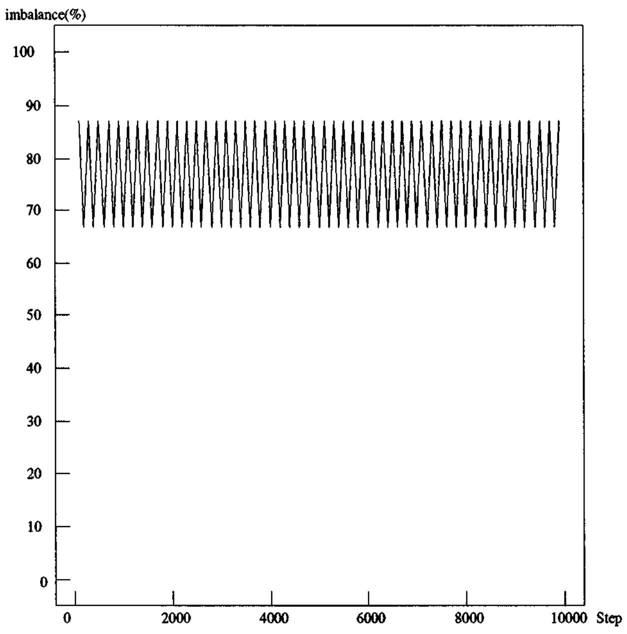


FIG. 6. Test 2 for Problem I. Load imbalance ratio vs MD step. Imbalance case, 10,000 MD steps.

problems when ϵ was set to 0.1. Almost perfect balance was achieved in approximately 20 iterations. This is shown in Fig. 5. Hence the parameter $S_{\max}$, which is the maximum number of balance steps allowed per balance routine call, was set to 20. I_g^{TOL} is the total amount of load imbalance necessary to actually do a load balance routine call. In our implementation an attempt to run the load balance routine is made every fixed number of MD steps. If

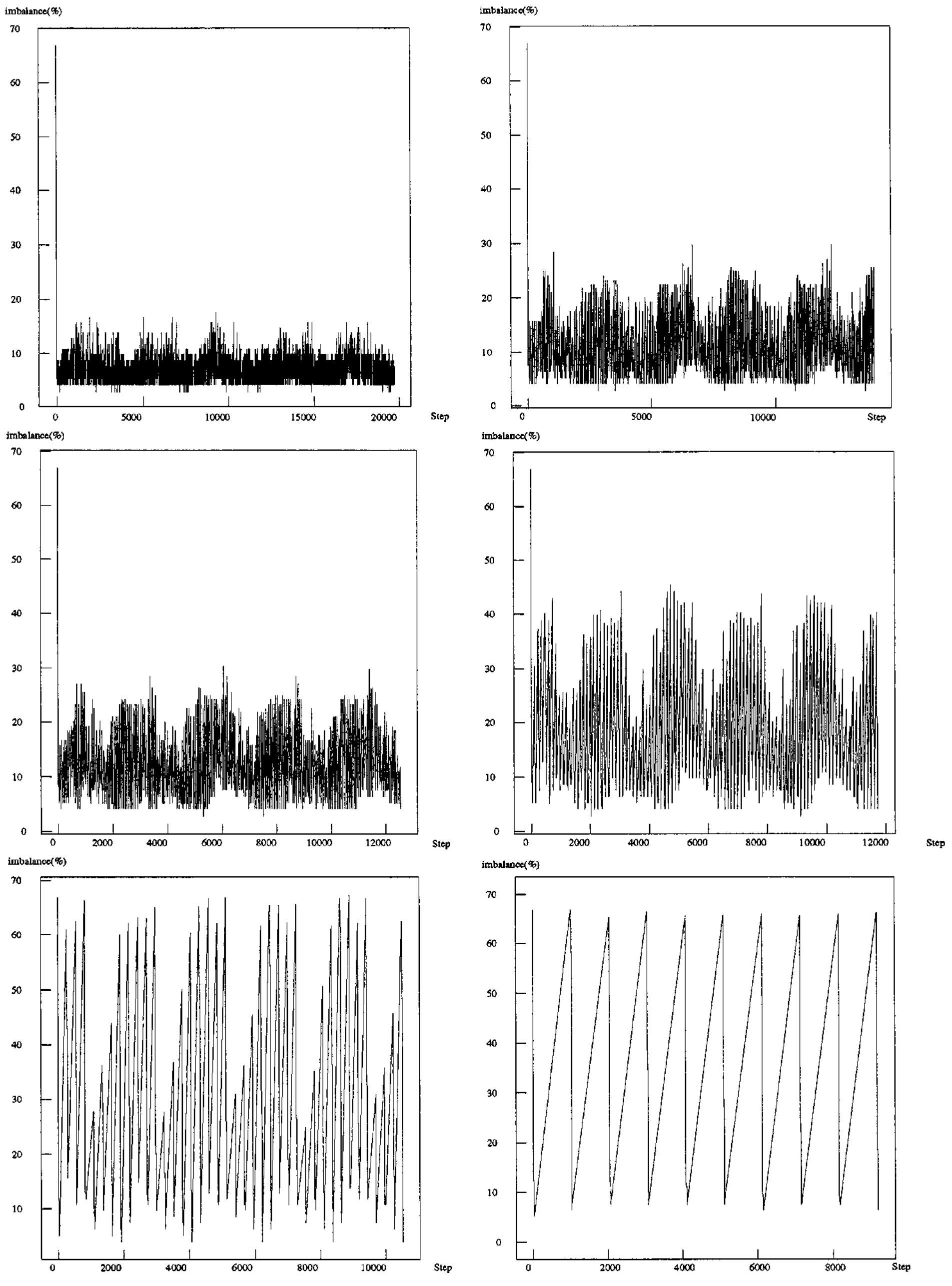


FIG. 7. Six tests for Problem I. Load imbalance ratio vs generic steps which include the usual MD steps and the load balance calls. In all tests, 10,000 MD steps are mixed with some load balance calls. Each load balance call allows up to 20 iterations. The figures, ordering from left to right and then top to bottom, show the results with one load balance call for every 10, 40, 50, 100, 250, and 1000 MD steps.

imbalance is less than I_g^{TOL} then no balance routine call is performed and the flow control returns to the MD calculation. Whenever the balance routine is called the maximum number of balance steps, as described in the algorithm description in Section 2, $S_{\max}$ is performed unless the load imbalance ratio falls below I_g^{TOL} in which case control returns to the MD calculation. I_l^{TOL} is a parameter used during the balance computation to decided whether a force vector is set to zero, i.e., to actually perform a particular vertex move. If local load imbalance of a set of four (2D) or eight (3D) subdomains is less than I_l^{TOL} the corresponding force vector is set to zero. The values used for the parameters were obtained form Test 1 whose imbalance ratio curve is shown in Fig. 5.

In Figs. 6 and 7, we show dynamic results for different frequencies of load balance calls.

In Fig. 8, we plot the computation time for different load balance experiments relative to the time for the unbalanced case. The number labeling each curve is the number of MD steps between load balance attempts for that computation. We note the oscillations, and we observe that they originate from the oscillatory behavior of the load imbalance, the reason being that particles are bouncing from the walls, moving together in a periodic fashion. Clearly, the more often one balances the better, up to a certain point, where balancing itself costs more than it gains. Therefore an optimal frequency exists for a particular problem. It can be seen in the plot that from among the curves obtained the best result is for 40 MD steps between load balance applications. For 10 MD steps between load balance the performance is already worse. However, it is interesting to see that while 50 MD/lb is worse than 40 MD/lb it is however also worse than the 100 MD/lb case. This can be explained by the fact that the load imbalance is periodic. Hence one would also expect that the performance is not a monotone function of the balancing frequency (although it seems to have a global optimal value).

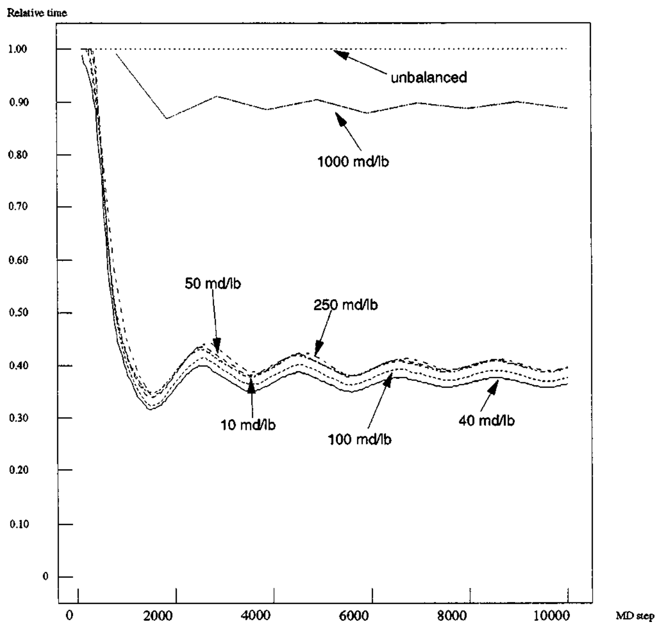


FIG. 8. Relative times versus number of generic steps for Problem I. The different curves are for different frequencies of load balancing. The numbers labeling each curve are the number of MD steps between load balance attempts.

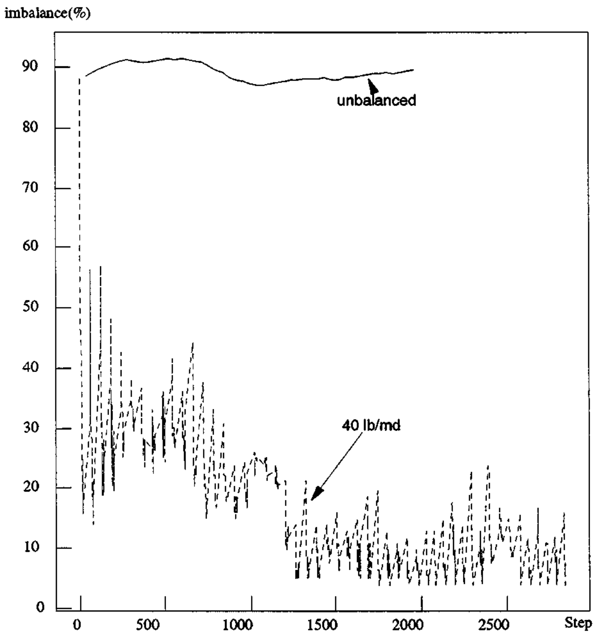


FIG. 9. Tests 1 and 2 for Problem II. Load imbalance ratio vs generic step. Imbalance case and balance every 40 steps. 10,000 MD steps. 20 balance steps max.

It is also interesting to see that the performance does not really change that much with frequency. A frequency of 40 MD/lb is only slightly better than the worst case shown at 250 MD/lb. The difference in timing is about 9% for 10,000 MD steps. A first rough analysis of the algorithm shows that while the MD computation takes about $N(N+1)/P$ steps, balancing takes of the order of N/P steps. Hence once one balances enough the computational time is dominated by the MD calculation and not the balancing, which explains the aforementioned insensitivity to balancing frequency.

Problem II. The domain is decomposed to $3 \times 3 \times 3$ subdomains, containing 2000 particles, with reflective boundary conditions. The MD computation is now done fully. We put a very heavy particle in the center of the domain, a cloud of 1999 lighter particles orbiting it. The force is an inverse square law. The initial angular distribution of particles is not uniform, but concentrates on a particular solid angle region, hence creating a high imbalance. This region circles the large particle periodically. Imbalance is also guaranteed by the fact that the distribution is spherical and the domain is a cube; hence the processors on the boundary always get fewer particles than the central processor. The imbalance ratio is high (about 90%) and the imbalance algorithm is shown to succeed in Fig. 9.

5. CONCLUSION

The performance of load balance algorithms is usually highly problem dependent. There have been other tiling approaches, but few are applicable to 3D problems. Recursive bisection is an approach which has been carried out with some success, but the resulting topology depends on the bisection history and cannot always be used to easily follow dynamically varying loads.

This paper is meant to establish the principle of the method and to demonstrate that it can succeed in a variety of load distributions.

We believe that this method can be used in a variety of similar problems, where some type of *load balance unit* (see [6]) exists or can be defined. For example, in finite difference methods where mesh refinement leads to an unbalanced load, one could subdivide the mesh into square blocks or units which are then transferred among processors to improve load balance. Of course, this implies that changes in current implementations must be made to adapt the particular problem to the load balance scheme. In finite difference methods load balance issues can also arise when an adaptive triangular grid generation method is implemented, so that different regions have different densities of grid points.

The method can also be extended to higher dimensions, with tetrahedra being replaced by the appropriate higher order simplices. However, with increasing dimension, the number of processors needed for even a very coarse grid rapidly becomes prohibitive.

ACKNOWLEDGMENT

The authors thank the National Science Foundation (Grant DMS9626859) for supporting this project.

REFERENCES

1. P. R. Eiseman, Adaptive grid generation, *Compt. Methods Appl. Mech. Eng.* **64**, 321 (1987).
2. Y. Deng, R. McCoy, R. Marr, R. Peierls, and O. Yasar, Molecular dynamics on distributed-memory MIMD computers with load balancing, *Appl. Math. Lett.* **8**, 37 (1995).
3. Y. Deng, R. McCoy, R. B. Marr, and R. F. Peierls, An unconventional method for load balancing, in *Proceedings, 7th SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, CA*, edited by D. Bailey *et al.* (1995), pp. 645–651.
4. P. R. Eiseman, Adaptive grid generation by mean value relaxation, *ASME J. Fluids Eng.* **107**, 477 (1985).
5. S. J. Plimpton, Fast parallel algorithms for short-range molecular dynamics, *J. Comput. Phys.* **117**, 1 (1995).
6. R. Alan McCoy and Y. Deng, Parallel particle simulations of thin-film deposition, *Int. J. High Perf. Comput. Appl.* **13**, 16 (1999).
7. M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids* (Oxford Univ. Press, London, 1987).
8. W. Hoover, *Molecular Dynamics*, Lecture Notes in Physics, Vol. 258 (Springer-Verlag, Berlin, 1986).
9. R. Hockney and J. Eastwood, *Computer Simulation Using Particles* (Inst. of Phys., Bristol, 1998).
10. J. Haile, *Molecular Dynamics Simulation* (Wiley–Interscience, New York, 1992).