



# Revisiting and parallelizing SHAKE

Yael Weinbach<sup>a</sup>, Ron Elber<sup>b,\*</sup>

<sup>a</sup> *The Hebrew University, Department of Computer Science, Givat Ram, Jerusalem 91904, Israel*

<sup>b</sup> *Cornell University, Department of Computer Science, Ithaca, NY 14853, USA*

Received 17 September 2004; received in revised form 16 March 2005; accepted 16 March 2005

Available online 11 May 2005

---

## Abstract

An algorithm is presented for running SHAKE in parallel. SHAKE is a widely used approach to compute molecular dynamics trajectories with constraints. An essential step in SHAKE is the solution of a sparse linear problem of the type  $Ax = b$ , where  $x$  is a vector of unknowns. Conjugate gradient minimization (that can be done in parallel) replaces the widely used iteration process that is inherently serial. Numerical examples present good load balancing and are limited only by communication time.

© 2005 Elsevier Inc. All rights reserved.

*Keywords:* Dynamics with constraints; Conjugate gradient; Lagrange multipliers

---

## 1. Introduction

Parallelization of a single Molecular Dynamic trajectory is one of the most promising approaches to extend the time scale of simulations. Here, we focus on distributed memory machines that are cheaper overall and have significant potential for further growth. In a Molecular Dynamic simulation the coordinates (and velocities) are propagated *serially* in small time steps following Newton's equation:

$$M\ddot{X} = -\nabla U. \quad (1)$$

The mass matrix is  $M$  (diagonal in the Cartesian representation) and  $U$  is the potential energy. If constraints are introduced  $\{\sigma_i(X) = 0\}_{i=1}^{\ell}$  (the total number of constraints is  $\ell$ ) then the equations of motions are modified to include Lagrange multipliers,  $\lambda_i$ , and constraint forces,  $-\nabla\sigma_i$

---

\* Corresponding author. Tel.: +607 255 7416; fax: +607 255 4428.

E-mail address: [ron@cs.cornell.edu](mailto:ron@cs.cornell.edu) (R. Elber).

$$M\ddot{X} = -\nabla U - \sum_l \lambda_l \nabla \sigma_l \quad (2)$$

subject to the constraints  $\sigma_l(X) = 0 \quad \forall l$ .

Calculations of the forces can take more than 90% of the computation time (if run serially). The estimates are based on the MOIL program [8] (see Section 8), though similar numbers are expected for other programs as well. If the fast degrees of freedom (e.g., bonds) are constrained with the SHAKE algorithm [18] a larger time step, typically by a factor of 2–3, can be used. The gain in computational time comes with the (small) penalty of 5–10% of serial calculation time. The simulations with SHAKE can therefore be made significantly longer in time at a similar computational cost to simulations of unconstrained dynamics.

Parallel and distributed computing is an attractive way to accelerate the simulations further. Indeed, significant advances were made in algorithms that compute Molecular Dynamic simulations in parallel. Implementations of parallel algorithms for Molecular Dynamic simulations focus on the calculation of the forces [4,5,7,10–13,15,16,19]. In the applications presented in this manuscript, atom decomposition is used. We call atom decomposition to algorithms that keep a complete copy of the coordinate and velocity vectors on each of the processors and distributes the calculation of the forces. It is especially suitable for systems that are not very large which we consider here (spatial decomposition, an alternative parallelization approach that divides the atoms between the processors, can be done efficiently for large systems).

Efficient algorithms to parallelize the calculations of the forces in the atomic or spatial decomposition schemes are available. However, parallelization of SHAKE is considerably more difficult to perform efficiently. The usual implementation of SHAKE (which we call bond relaxation [18]) is inherently serial and cannot be parallelized in the general case. The parallelization of direct relaxation methods – one bond is corrected at a time, are difficult to implement and to exactly reproduce the serial algorithm. Therefore practical applications are either approximate, or the algorithms are specifically tailored to the problem at hand. The complexity of the parallelization of SHAKE is related to the constraints' coupling matrix  $A_{ll'}$  to be defined below (Eq. (11)). Two constraints,  $l$  and  $l'$ , are *directly* coupled if  $A_{ll'} \neq 0$ , and coupled if an integer  $n$  exists such that  $(A^n)_{ll'} \neq 0$ . Significant direct coupling makes the parallel (and serial) implementation of SHAKE more difficult. Special cases in which the matrix  $A_{ll'}$  decomposes into small blocks (e.g., water molecules) can be parallelized in a trivial way. In fact, our algorithm exploits block structure of the coupling matrix, for example water molecules for which the usual SHAKE of bond relaxation converges more slowly than our algorithm. Moreover, for complex polymer simulations (such as protein folding), parallel decoupling or exact bond relaxation are not possible. These cases become more important as polymer simulations without explicit solvent gain in popularity (for implicit solvent papers see for instance the generalized Born model [9,21,24]). Examples for previous heuristic approaches are as follows: Debolt and Kollman [7] introduced a pipeline procedure for relaxation of bond lists, and Brown et al. [3] introduced a multi-coloring scheme for sequential relaxation of disconnected “islands” of bonds to avoid shared bonds between “working” processors.

For massively parallel and memory-distributed systems it is essential to parallelize SHAKE in addition to the forces. A program can be partitioned into a part that can be made parallel and a part that must run serially. The maximal speedup is bound by the fraction of serial code that remains, and serial SHAKE can take up to 10% of the total execution time.

Parallelization of SHAKE is challenging *not* because of the requirement for load balancing, because for SHAKE this is static, can be done prior to the beginning of the simulation and is relatively easy to do. One difficulty is that the usual SHAKE formulation requires *serial* adjustment of constraints (bonds), one bond after the other. An alternative to the bond relaxation algorithm is therefore desirable. Another difficulty is that the serial algorithm requires numerous iterations. When the forces are computed, they are computed once in a time step. In contrast, usual SHAKE bond relaxations may iterate numerous times in a single time

step and therefore require closing and opening a number of communication channels (the actual amount of data transfer is not so large). As part of an effective parallelization of SHAKE it is desirable to reduce the number of iterations as much as possible, and to make the algorithm rigorously equivalent to the serial version. Finally if a true parallel algorithm for constrained dynamics is formulated, the problem of communication overhead remains. While this overhead diminishes as the system size increases for the (small) examples of the present manuscript the communication overhead is significant. We have put forward the idea that a matrix formulation of SHAKE (known already from the original work of Ryckaert and Ciccotti [18] and further developed in the work of Barth et al. [2]) in conjunction with the conjugate gradient algorithm is an effective way of addressing the above points. For enhanced efficiency the conjugate gradient algorithm is using a symmetric version of the matrix that was introduced by Barth et al. [2]. Of course it is important for the new algorithm to exploit a potential block structure of  $A_{ll'}$ , which is also what we have done. The algorithm for parallelization of SHAKE that we present here is expected to work for both atomic and spatial decomposition, even though we have pursued here (in accord with our research interest) only atom decomposition, which is more appropriate for small systems. Further evaluation of the spatial decomposition option can be found in Section 9.

Another comment is about hardware architecture. We restrict the numerical examples to the (cheaper) solution of distributed-memory parallel-systems such as clusters (as opposed to shared memory computers). Although the same algorithm can be effective for both architectures, the distributed memory system is harder to optimize.

Below we present a brief overview of constraint dynamics and then discuss the proposed parallel algorithm for SHAKE. We close by numerical examples.

## 2. Constrained dynamics and SHAKE

The beauty of the SHAKE algorithm is the realization that the constraints need to be satisfied exactly at each step, and that it is insufficient to determine the exact Lagrange's multipliers of the differential equation [18]. It is necessary to determine the coordinates that satisfy the constraints exactly for a finite time step. Alternatively, (as is done below) the Lagrange multipliers can be defined in the context of a finite time step. If only the constraint forces and Lagrange's multipliers from the differential equation are used, the numerical errors in the constraints (using a finite integration step  $\Delta t$ ) accumulate during the simulation. By the end of the day and the end of a long simulation, very significant errors in the constraints are evident. It is necessary to consider explicitly the equations for the constraints (and not only the constraint forces) when integration with a finite step is used.

At this point it is convenient to introduce a concrete numerical algorithm, and we choose the velocity form of the Verlet algorithm [23]

$$\begin{aligned} X_{n+1} &= X_n + V_n \Delta t + \frac{\Delta t^2}{2} M^{-1} \left( -\nabla U_n - \sum_{l'} \lambda_{nl'} \nabla \sigma_{nl'} \right), \\ V_{n+1} &= V_n + \frac{\Delta t}{2} M^{-1} \left( -\nabla U_n - \sum_{l'} \lambda_{nl'} \nabla \sigma_{nl'} - \nabla U_{n+1} - \sum_{l'} \lambda_{n+1l'} \nabla \sigma_{n+1l'} \right). \end{aligned} \quad (3)$$

We denote by  $X_n$  and  $V_n$  the coordinate and the velocity vectors at step  $n$ . Below we deal first with the constraints on the coordinates. The velocity constraints that are simpler to handle will come later. Constraining explicitly the velocities is also called RATTLE [1]. Lagrange's multipliers that exactly satisfy the constraints are not known (i.e.,  $\sigma_l(X_{n+1}) = 0 \forall l$ ) and need to be determined from the equations of the constraints.

We denote by  $X_{n+1}^{(0)}$  and  $V_{n+1}^{(0)}$  the vectors generated from the  $X_n$  and  $V_n$  vectors by a single integration step in the *absence of the constraints*. The constraint at  $X_{n+1}$  is expanded linearly near  $X_{n+1}^{(0)}$

$$\begin{aligned}\sigma_l(X_{n+1}) &= 0 \simeq \sigma_l(X_{n+1}^{(0)}) + \nabla\sigma_l|_{X=X_{n+1}^{(0)}}(X_{n+1} - X_{n+1}^{(0)}) \\ &= \sigma_l(X_{n+1}^{(0)}) - \nabla\sigma_l|_{X=X_{n+1}^{(0)}} \frac{\Delta t^2}{2} M^{-1} \sum_{l'} \lambda_{l'} \nabla\sigma_{l'}|_{X=X_n}.\end{aligned}\quad (4)$$

We use Eq. (3) to record the difference between  $X_{n+1}$  and  $X_{n+1}^{(0)}$  as  $\frac{\Delta t^2}{2} M^{-1} \sum_{l'} \lambda_{l'} \nabla\sigma_{l'}$ . The unknowns in Eq. (4) are Lagrange's multipliers  $\lambda_{l'}$ . We define the matrix  $A_{ll'}^{(0)}$

$$A_{ll'}^{(0)} = \nabla\sigma_l|_{X=X_{n+1}^{(0)}} \frac{\Delta t^2}{2} M^{-1} \nabla\sigma_{l'}|_{X=X_n} \quad (5)$$

and write in a more compact form a set of linear equations for  $\lambda_{l'}$

$$\sigma_l(X_i^{(0)}) = \sum_{l'} A_{ll'}^{(0)} \lambda_{l'}. \quad (6)$$

The solution of these linear equations is a major bottleneck of the calculations in the SHAKE algorithm. We denote the solution by  $\lambda_{l'}^{(0)}$ . Since the constraints are, in general, nonlinear, Eq. (4) is approximate and the exact solution of the linear equation is insufficient. We construct a first-order correction to the exact coordinates  $X_{n+1}$

$$X_{n+1}^{(1)} = X_{n+1}^{(0)} - \frac{\Delta t^2}{2} M^{-1} \sum_{l'} \lambda_{l',n}^{(0)} \nabla\sigma_{l',n}. \quad (7)$$

This intermediate coordinate vector is plugged back into the nonlinear equations of the constraints. A new linear expansion of the constraint (now near  $X_{n+1}^{(1)}$ ) interpolates to the exact coordinate  $X_{n+1}$ , and defines a Newton's-like procedure for the iterative solution of a nonlinear equation by a sequence of linear approximations

$$\begin{aligned}\sigma_l(X_{n+1}) &= 0 \simeq \sigma_l(X_{n+1}^{(1)}) + \nabla\sigma_l|_{X=X_{n+1}^{(1)}}(X_{n+1} - X_{n+1}^{(1)}) \\ &= \sigma_l(X_{n+1}^{(1)}) - \nabla\sigma_l|_{X=X_{n+1}^{(1)}} \frac{\Delta t^2}{2} M^{-1} \sum_{l'} (\lambda_{l',n} - \lambda_{l',n}^{(0)}) \nabla\sigma_{l'}|_{X=X_n}.\end{aligned}\quad (8)$$

The expression for  $X_{n+1}^{(1)}$  from Eq. (7) is used in Eq. (8). Note the similarity to Eq. (4) that is our zero-order iteration. The difference between Eqs. (4) and (8) (Eq. (8) is our first-order iteration) is the use of the update of Lagrange's multiplier. We define  $\lambda_{l',n}^{(1)} \equiv (\lambda_{l',n} - \lambda_{l',n}^{(0)})$  and a new matrix that couples the constraints  $A_{ll'}^{(1)} = \nabla\sigma_l|_{X=X_{n+1}^{(1)}} \frac{\Delta t^2}{2} M^{-1} \nabla\sigma_{l'}|_{X=X_n}$ . A linear equation for  $\lambda_{l',n}^{(1)} - \sigma_l(X_i^{(1)}) = \sum_{l'} A_{ll'}^{(1)} \lambda_{l',n}^{(1)}$  is formulated. The solution,  $\lambda_{l',n}^{(1)}$ , enables us to write the next approximation

$$X_{n+1}^{(2)} = X_{n+1}^{(1)} - \frac{\Delta t^2}{2} M^{-1} \sum_{l'} \lambda_{l',n}^{(1)} \cdot \nabla\sigma_{l',n}|_{X=X_n} = X_{n+1}^{(0)} - \frac{\Delta t^2}{2} M^{-1} \sum_{l'} (\lambda_{l',n}^{(1)} + \lambda_{l',n}^{(0)}) \cdot \nabla\sigma_{l',n}|_{X=X_n}. \quad (9)$$

And in more general terms

$$X_{n+1}^{(k)} = X_{n+1}^{(k-1)} + \frac{\Delta t^2}{2} M^{-1} \sum_{l'} \lambda_{l',n}^{(k-1)} \nabla\sigma_{l',n} = X_{n+1}^{(0)} + \frac{\Delta t^2}{2} M^{-1} \sum_{l'} \left( \sum_{k'=1}^{k-1} \lambda_{l',n}^{(k')} \right) \nabla\sigma_{l',n}. \quad (10)$$

Note that in order to determine  $\lambda_{l',n}^{(k)}$  we need to reconstruct the matrix  $A_{ll'}^{(k)}$ . The iterations terminate when  $|\sigma_l(X_{n+1}^{(k+1)})| \leq \varepsilon \forall l$ , where  $\varepsilon$  is a small positive number.

### 3. Further discussion of the matrix $A$

In most applications  $A$  is a highly sparse matrix, reflecting a few weakly coupled constraints. Constraining the bonds in a polymer is a typical case. For example,  $\sigma_l = (r_i - r_j)^2 - d_{ij}$ , where  $r_i$  and  $r_j$  are the vector positions of atoms  $i$  and  $j$ , and  $d_{ij}$  is their fixed (constrained) distance. Since a single atom typically has 2–4 bonds, the matrix is highly sparse, suggesting that a greedy algorithm [6] is likely to be successful. Indeed, a serial algorithm was designed for fixing the coordinates along the directions determined by  $\nabla\sigma_l$ , adjusting the coordinates according to one constraint at a time. This algorithm avoids the inversion of the matrix  $A$  and the explicit calculation of Lagrange multipliers. The above algorithm is, however, inherently serial and is difficult to parallelize in a consistent and rigorous way for the reasons discussed in the introduction. It is also limited to highly sparse matrices.

We therefore return to the formulation in (6). The Lagrange multipliers could be found using standard linear techniques, as discussed in the original paper of SHAKE [18]. Use of the linear equations, and a solution of the inverse of the matrix  $A_{ll'}$  on a shared memory machine were discussed in Mertz et al. paper [11]. Here, we exploit further the matrix formulation. In particular, we note that usually a reasonable guess for Lagrange's multipliers is available (e.g., Lagrange's multipliers from the previous step). It therefore makes sense to build on this guess and to determine Lagrange's multipliers not by straightforward solution of linear equations but using minimization based on the initial guess. The conjugated gradient algorithm comes to mind here. What function should we minimize? In principle any linear equation of the form  $Ax = b$  could be re-written in a form ready for conjugate gradient minimization as  $F(x) = (Ax - b)^t(Ax - b) = 0$ . However, this formulation is expected to be more expensive than the approach we finally used (see below) for two reasons: first, we need to compute matrix multiplication (an  $\ell^3$  operation in a worst case scenario), and second, we need to re-compute the matrix in every SHAKE iteration. These two conditions led us to use the symmetric matrix formulation [2]. We note that the difference between  $\nabla\sigma_l|_{X=X^{(0)}}$  and  $\nabla\sigma_l|_{X=X_i}$  is of order of  $O(\Delta t)$ . Ignoring this difference makes our original linear expansion correct to the order of  $O(\Delta t^3)$  instead of  $O(\Delta t^4)$  and simplifies the matrix  $A_{ll'}^{(k)}$  to a symmetric form

$$A_{ll'}^{(k)} \cong \nabla\sigma_l|_{X=X_i} \frac{\Delta t^2}{2} M^{-1} \nabla\sigma_{l'}|_{X=X_i} \equiv A_{ll'}. \quad (11)$$

We are losing one-order of accuracy ( $O(\Delta t^3)$  instead of  $O(\Delta t^4)$ ), which means that the calculations employing  $A_{ll'}$  will converge more slowly (a larger number of iterations) and the radius of convergence may be smaller. In practice, the number of iterations increases by 10–20%, and no change in the radius of convergence was observed.

Despite the above disadvantages it is efficient to use the symmetric matrix, and to minimize an alternative function  $F(x) = 1/2x^tAx - x^tb$  because:

- (a) The non-symmetric matrix,  $A_{ll'}^{(k)}$  is re-computed each of the  $k$  iterations described in the previous section. The symmetric matrix,  $A_{ll'}$ , is computed only once for all the iterations associated with a single time step.
- (b) We avoid the need to compute matrix multiplications in the general implementation of conjugate gradient for asymmetric matrices.

We also need the matrix  $A_{ll'}$  to be positive definite. For a matrix to be positive definite it is sufficient to show that for any real non-zero vector  $a$  of length  $\ell$  ( $\ell$  is the number of constraints) we have  $a^tAa > 0$ . The matrix  $A_{ll'}$  can be represented as a sum of scalar products  $y_l$  and  $y_{l'}$  ( $y_l = t \frac{\Delta t}{\sqrt{2}} \cdot M^{-1/2} \cdot \nabla\sigma_l$ ). The matrix element is

$$a^t A a = \sum_{l'} a_l A_{ll'} a_{l'} = \sum_{l'} a_l \left( \sum_k y_{lk} y_{l'k} \right) a_{l'} = \sum_k \left( \sum_l a_l y_{lk} \right) \left( \sum_{l'} a_{l'} y_{l'k} \right) = \sum_k \left( \sum_l a_l y_{lk} \right)^2 > 0. \quad (12)$$

In Appendix A, we consider a special case of the matrix  $A_{ll'}$ . This special case is of considerable practical interest since it requires only a *single* calculation of the matrix throughout the simulation.

#### 4. Determination of the Lagrange multipliers by minimization

For a positive definite matrix it is possible to efficiently solve  $\lambda_{l'}$  by a minimization procedure (here we used conjugate gradient with pre-conditioning [14]). We define a quadratic target function (in  $\lambda_{l'}$ ) with a minimum at the desired solution for Lagrange's multipliers

$$F = \frac{1}{2} \lambda^t A \lambda - \lambda^t \sigma. \quad (13)$$

Given that  $A$  is positive definite the minimization of  $F(\frac{dF}{d\lambda} = A\lambda - \sigma = 0)$  is equivalent to the linear problem stated above, which clarifies why we insisted in the previous section on having  $A$  symmetric and positive definite. (We note that there is another way to find  $\lambda$ ; as we discussed in the previous section, it is also possible to use non-symmetric  $A$  using minimization of a quadratic function in  $\lambda$ . Consider  $\Delta = \sigma - A\lambda$  and a target function  $F' = \Delta^t \cdot \Delta$ .  $F'$  is more complex to compute and includes (the expensive) computations of  $A^t A$ . We therefore did not pursue this alternative in our studies.)

For completeness and to appreciate the parallelization process, we briefly describe the conjugate gradient algorithm. In conjugate gradient we perform a sequence of line minimizations. For that purpose we need to determine the direction of the current line minimization (for example, in the steepest descent minimization the direction is the opposite direction of the gradient of the target function). The second ingredient is to determine the size of the displacement in the predetermined direction. The two steps are defined:

- (a) The determination of  $p_k$ , the vector of the direction of search at step  $k$ .
- (b) The determination of the step size,  $\alpha_k$ , a scalar that determines the size of the displacement along the  $p_k$  direction in step  $k$ .

We define the gradient of the function  $F$  at minimization step  $k$  as  $g_k \equiv A\lambda_k - \sigma_k$ . In the first step the search exactly follows the direction of the gradient  $p_1 = -g_1$ . The minimum position along the line defined by  $p_1$  and the initial value  $\lambda_0$  is determined by a single parameter  $\alpha$ :  $\lambda_1 = \lambda_0 + \alpha p_1 = \lambda_0 - \alpha g_1$ . The gradient of the function at the minimum along the search line must be orthogonal to  $g_1$ . We therefore have:  $g_1 \cdot [A(\lambda_0 + \alpha p_1) - \sigma] = 0 \rightarrow -\alpha g_1^t A g_1 + g_1^t g_1 = 0$  and  $\alpha_1 = \frac{g_1^t g_1}{g_1^t A g_1}$ . The above scheme describes a steepest descent step, which is used in the conjugate gradient algorithm only for the first time. For any other minimization step  $k$  ( $k \neq 1$ ) the conjugate gradient search direction is  $p_k = -g_k + \beta_k \cdot p_{k-1}$ . The parameter  $\beta_k$  is chosen to ensure that the new search direction ( $p_k$ ) is “conjugate” to the previous search direction ( $p_{k-1}$ ), which means satisfying the condition  $p_k A p_{k-1} = 0$ . This condition ensures that sequential minimizations are in independent directions for quadratic functions, and convergence to the exact answer by at most  $\ell$  steps ( $\ell$  is the number of Lagrange's multipliers). In practice the number of steps can be reduced significantly [14]. In our experience only a few minimization steps (about four) are required to find high-quality Lagrange multipliers for hundreds to thousands of constraints. The scalar parameter  $\beta_k$  is determined by  $\beta_k = \frac{g_k^t g_k}{g_{k-1}^t g_{k-1}}$ , the step size in the new search direction is  $\alpha_k = \frac{g_k^t g_k}{p_k^t A p_k}$ , and the new Lagrange's multipliers are  $\lambda^{(k+1)} = \lambda^{(k)} + \alpha_k p_k$ .

One more way of enhancing the performance of conjugate gradients is the use of preconditioning. To accelerate the convergence, an approximate inverse can be helpful. Consider the linear equation of interest  $A\lambda = b$  and let  $B$  be non-singular matrix. The more degenerate the eigenvalue spectrum of  $A$ , the more efficient the conjugate gradient algorithm. Define  $\eta = B\lambda$ , we then have  $((B^{-1})^t A B^{-1})\eta = (B^{-1})^t b$ , which depending on the properties of  $B$  can accelerate the convergence significantly. We have used the diagonal of  $A$  to define a diagonal matrix  $B = (A^D)^{1/2} = [\text{diagonal}(A)]^{1/2}$  with a trivial inverse. This simple preconditioning squeezed another iteration of two on our behalf. We define  $C \equiv B^t B$ ,  $\varepsilon$ -maximum allowed error,  $k_{\max}$  maximum number of iterations, and outline the algorithm below

Input:  $\lambda_1$ , and  $A$ ,  $\sigma$ ,  $C$

Initiate loop:

$$k = 0,$$

$$g_1 = A\lambda_1 - \sigma.$$

$$\eta_1 = C^{-1}g_1,$$

$$p_1 = -g_1.$$

For ( $|g_{kt}| \geq \varepsilon$ ,  $k = k + 1$ ,  $k < k_{\max} - 1$ )

$$\alpha_k = \frac{g_k^t \eta_k}{p_k^t A p_k},$$

$$\lambda_{k+1} = \lambda_k + \alpha_k p_k,$$

$$g_{k+1} = g_k + \alpha_k A p_k,$$

$$\eta_{k+1} = C^{-1}g_{k+1},$$

$$\beta_{k+1} = \frac{g_{k+1}^t \eta_{k+1}}{g_k^t \eta_k},$$

$$p_{k+1} = -\eta_{k+1} + \beta_{k+1} p_k.$$

(14)

end

The computational efforts of the conjugate gradient algorithm, preconditioned or not, are focused on the matrix vector multiplication  $A p_k$  and the scalar products  $g_{k+1}^t \cdot \eta_{k+1}$ .

Alternatives for solving the linear equations are possible (for example, Gaussian elimination [22]), and we did not explore them in detail. Nevertheless, the conjugate gradient iterations converge quickly and parallelize easily, as described below, making it a useful technique for the problem at hand.

## 5. Representation of the sparse matrix $A$

The brief description of the minimization algorithm shows that in each iteration step, we need to perform matrix–vector multiplication and a few scalar products. Since the matrix ( $A$ ) is sparse we wish to represent it in a way that (a) will make it possible to perform efficient matrix–vector multiplication and (b) will store only non-zero elements. There are many representations of sparse matrices that can be used in principle. We have chosen one representation that works well for the task at hand. While we have not made a careful investigation of compact representation of sparse matrices in the context of the application at hand we looked at a few alternatives. The requirements of compact storage, minimizing communication between processors, and ease of access to relevant elements are not trivial to satisfy simultaneously and work well

in the protocol discussed below. Two vectors are stored: (i) a vector,  $V$ , with all the elements of  $A$  that are not zero and (ii) a pointer,  $P$ , to these elements. The first  $\ell$  elements of  $V$  are the diagonal elements of  $A$  and the off-diagonal elements follow, one row at a time. The first  $\ell$  elements of the vector  $P$  point to the end of the rows of the matrix  $A$  as stored in the vector  $V$ . The element  $\ell + k$  of  $P$  is the column in  $A$  of the  $\ell + k$  element of  $V$ . For example, if the sparse matrix  $A$  is

$$A = \begin{pmatrix} 1.2 & 4.1 & 0 & 0 \\ 4.1 & 7.9 & 0 & 0 \\ 0 & 0 & 9 & 0 \\ 0 & 0 & 0 & 8.2 \end{pmatrix},$$

we have  $V = (1.2 \ 7.9 \ 9 \ 8.2 \ 4.1 \ 4.1)$  and  $P = (5 \ 6 \ 6 \ 6 \ 2 \ 1)$ .

The connectivity of the system is not changing during the simulation. We do not add, or remove constraints or change the identity of the atoms that are constrained. The vector  $P$  is therefore fixed during the simulation. For each time step, only the vector  $V$  (the non-zero elements of the matrix  $A$ ) requires an update. The number of these elements is much smaller than the size of the matrix  $A$ . For example, the number of bond constraints in myoglobin (a numerical example that we present below) is 1589. The length of the vector  $V$  is 6174, significantly smaller than  $1589 \times 1589$  (99.7% of the elements are zero).

## 6. Parallelization of conjugate gradient

The list of constraints is divided between all the processors. Each processor has a fixed number of constraints that it works on, and that does not change during the simulation. There are three types of operations that require parallelization:

- (1) *An inner-product vector.* Each processor multiplies the elements from its own list and sums them up. A summation of the results from the different processors is performed.
- (2) *Summation of vectors to find the spatial displacement vector (Eq. 10).* Every processor sums up the contribution to the displacements arising from its own set of constraints and the contributions to the displacements are summed over all processors.
- (3) *Multiplication of a vector by a matrix.* This is the most complex part. The type of parallelization that we used assumes that the matrix  $A$  is sparse with most of the elements centered near the diagonal of the matrix. Each of the processors is responsible for only a part of Lagrange's multipliers vector and it saves only the rows in the matrix that directly multiply this part of the vector. Of course, to obtain a correct result for the vector matrix multiplication we need elements that are not available in the current processors. These elements are transferred primarily from nearby processors since the matrix elements are concentrated near the diagonal. Since our prime interest is in "almost" linear polymers (as proteins are), the natural ordering of atoms in the polymer follows (roughly) the diagonal of the matrix of constraints. We also note that uncoupled blocks of constraints (e.g., water molecules) are treated in the same way. We obviously make sure that a single water molecule (a block of three connected bonds) is not broken between different processors.

Nevertheless, highly off-diagonal constraints are also possible (for example between the side chains of cysteine residues in proteins). Therefore each of the processors has three lists of Lagrange's multipliers to send:



- (i) a list of elements to send to the nearby processor on the right,
- (ii) a list of elements to send to the nearby processor on the left,
- (iii) a list of elements to send to a processor that is not close.

Of course list (iii) is usually very small in proteins, making the assumption of “near the diagonal” sound. These elements are sent to the processors that need them prior to the multiplication. After the communication is complete the vector–matrix multiplication is performed independently on each of the nodes.

## 7. The velocity constraints

The velocity constraints are implemented as described in RATTLE [1]. We consider the constraint  $\sigma_l(X)$  and differentiate it with respect to time

$$\frac{d\sigma_l(X)}{dt} = \frac{d\sigma_l}{dX} \cdot V = \nabla\sigma_l \cdot V = 0 \quad \forall l. \quad (15)$$

Of course, we wish the above scalar product to be zero for the newly generated step  $X_{n+1}$ . Note that we consider Eq. (15) only after we computed the coordinates of the next step. We therefore have from Eq. (3)

$$\begin{aligned} V_{n+1} &= V_n + \frac{\Delta t}{2} M^{-1} \left( -\nabla U_n - \sum_{l'} \lambda_{nl'} \nabla \sigma_{nl'} - \nabla U_{n+1} - \sum_{l'} \lambda_{n+1l'} \nabla \sigma_{n+1l'} \right), \\ V_{n+1} &= \bar{V}_n + \frac{\Delta t}{2} M^{-1} \left( -\nabla U_{n+1} - \sum_{l'} \lambda_{n+1l'} \nabla \sigma_{n+1l'} \right) = V_{n+1}^{(0)} - \frac{\Delta t}{2} M^{-1} \left( \sum_{l'} \lambda_{n+1l'} \nabla \sigma_{n+1l'} \right), \\ \bar{V}_n &= V_n + \frac{\Delta t}{2} M^{-1} \left( -\nabla U_n - \sum_{l'} \lambda_{nl'} \nabla \sigma_{nl'} \right), \\ V_{n+1}^{(0)} &= \bar{V}_n + \frac{\Delta t}{2} M^{-1} (-\nabla U_{n+1}). \end{aligned} \quad (16)$$

Since the coordinates  $X_{n+1}$  are known before we propagate the velocities we can compute  $V_n^{(0)}$  directly, and we have for the constraints

$$V_{n+1} \cdot \nabla \sigma_{n+1,l} |_{X=X_{n+1}} = V_n^{(0)} \cdot \nabla \sigma_{n+1,l} |_{X=X_{n+1}} - \frac{\Delta t}{2} \sum_{l'} \lambda_{n+1,l'} \nabla \sigma_{n+1,l'} \cdot M^{-1} \nabla \sigma_{n+1,l} |_{X=X_{n+1}} = 0. \quad (17)$$

The matrix  $\frac{\Delta t}{2} \nabla \sigma_{n+1,l'} \cdot M^{-1} \nabla \sigma_{n+1,l}$  is  $A_{l,l'}/\Delta t$  for the time step  $n+1$ . The same matrix is therefore used for coordinates and velocities. The constraints (Eq. (15)) are linear in Lagrange’s multipliers. The linear equation (17) is therefore exact and no Newton’s-like iterations are required to determine Lagrange’s multipliers.

## 8. Numerical experiments

The computational experiments were performed on an SP2 computer using MPI. Two molecular systems were simulated: (i) the protein Myoglobin with 1563 atoms and 1589 bond constraints and (ii) a DMPC lipid bilayer with water solvation (9630 atoms and 7668 bonds). All the runs were performed with 11 Å cut-off distance for electrostatic and 8 Å for Lennard Jones interactions. The time step was 1 fs, and the temperature was 300 K. The non-bonded lists were updated every 10 steps.

The parallel SHAKE algorithm discussed above was implemented in the program MOIL [8]. For completeness we briefly summarize the complete parallelization of the program.

The parallelization is based on atom decomposition in which interaction lists are divided between the processors. Partial forces are computed on each of the processors and the vector of forces is gathered and redistributed to all of the processors to be used in a new integration step. Some of the lists of interactions are fixed (i.e., lists of bonds, angles, torsions, improper torsions and 1–4 interactions), and some are dynamically constructed (Lennard Jones and electrostatic lists). The division between the processors of the fixed lists is trivial and is done at the beginning of the simulation. The dynamically updated lists of the non-bonded interactions are computed in parallel as well and were demonstrated to yield excellent load balancing.

When the SHAKE algorithm was used we also parallelize the Verlet integrator. For the systems we investigate, the number of SHAKE iterations that were required never exceeded 4. The convergence criterion was  $10^{-7}$  Å accuracy for each of the bond lengths.

In Table 1, we present the result for the simulation of the protein myoglobin for 2000 steps of dynamics. This is a relatively small system for which the speedup is not expected to be significant. Nevertheless, the table shows some speedup (a factor of 2 for four processors) and good load balancing, when the individual routines are examined, suggesting the major problem to be the communication overhead for the SP2. Of course when the system is small, the communication problem becomes more critical. A larger molecule will make the ratio of computations to communication less overwhelming.

In Table 2, we consider a second example of membrane simulations with SHAKE. The larger system naturally gives better results. We note that the set of 9630 atoms is not exceptionally large. There are numerous studies of molecular systems of similar and larger sizes. The membrane system is easier to constrain since the connectivity is lower, making the constraint matrix not only sparse but also made of blocks. We did not make an explicit use of the block structure of the matrix, however, this clearly makes the calculations less demanding.

In Table 2, we summarize the run time of the parallel MOIL program with SHAKE. The speedup is bound primarily by communication time. It is expected that faster communication protocols (and the use of even larger systems) will improve the speedup factors. It also means the algorithm so presented is expected to be more efficient on shared memory machines in which communication overhead is moot. We comment that the usual serial version of SHAKE (the version that employs bond relaxation) is faster in our hands than a serial version of the matrix SHAKE (with the conjugate gradient minimization) by a factor of about 2 in the myoglobin system. Other matrix implementations seem to perform somewhat better than regular SHAKE [2]. However, our code was not optimized to that level. This observation means that to obtain a gain for a system like myoglobin (compared to the serial version of SHAKE) we need to run on more than 2 CPUs in parallel.

Another comment is concerned with load balancing. Load balancing can be divided into static and dynamic lists, where static lists do not change during the simulations and are easy to divide between the processors (e.g., bond or SHAKE lists). The dynamic lists in our code are those describing non-bonded interactions, and are in principle harder to maintain and to balance compared to the static lists. In MOIL we derive first a non-bonded list for residue-residue interactions and only then produce atom pair-lists. This intermediate step adds to the complications since parallelization, done on the residue level, is not necessarily well balanced. We are therefore using a self-correcting algorithm that “guesses” an optimal partitioning between the processors following the residue lists. These residue lists are corrected for future calculations by adjusting the residue selections according to the observed atomic lists. Despite the potential difficulties, the length of our non-bonded lists deviates only slightly from ideal partitioning as can be exemplified by the lengths of the observed atomic pair-lists in the myoglobin simulation. The atomic lists, after the first adjustment of the residue lists (using 4 processors), include 41999, 42373, 41876, and 42088 atom pairs. The deviation from ideal partition is less than 1%.

Table 1  
Time distribution of critical subroutines in 2000 steps of dynamics for the myoglobin system

| Function  | No. of processors |        |        |        |
|---|-------------------|--------|--------|--------|
|   | 1                 | 4      | 8      | 16     |
| cdie  | 271.81            | 274.16 | 292.64 | 294.08 |
| nbond   | 75.43             | 78.32  | 62.96  | 76.32  |
| mult_mat_vec  | 58.06             | 59.76  | 62.4   | 58.81  |
| build_mat_from_idx                                    | 7.88              | 11.64  | 16.96  | 28.32  |
| conjugate_grad  | 10.05             | 10.68  | 11.52  | 15.04  |
| mult_vec_vec  | 11.92             | 11.84  | 13.6   | 12.84  |
| Total SHAKE   | 87.91             | 93.92  | 104.48 | 115.01 |
| Total time in seconds (summation over all processors) | 489.99            | 946.4  | 1792   | 3712   |
| Speed up factors                                      | 1                 | 2.1    | 2.22   | 2.15   |

The time step in all simulations was 1 fs. Conjugate gradient represents all the other operations in the minimization that are not included in specific functions (for example, evaluation of the convergence). In theory the entry to the tables must be non-decreasing when moving from left to right. However, memory access is sometimes improved when the fraction of the system size that is studied in a single CPU is smaller and better fit into memory. The times provided (in seconds) are a sum over all processors. The summation can be estimated more accurately than the total clock time, and since the load balancing is excellent (see text) the sum is similar for each of the processors. Note the excellent results for individual routines and the less impressive results for the total time. A major factor contributing to the total time (that does not show up in the individual routines) is the communication overhead. On the system reported below (of a relatively small system, the isolated myoglobin molecule – 1563 atoms) the speedup on four processors comparing serial and parallel versions without SHAKE was 3.5, a similar comparison with SHAKE yielded a factor of 2.1. Since SHAKE allows doubling of the time step there is still a gain on four processors. The speedup for the membrane simulation with SHAKE (a system that is moderately large – 9630 atoms – see Table 2) was 11 on 16 processors. The same time step (1 femtosecond) was used in all the simulations. A detailed explanation of the different routines follows:

- (i) *cdie* – The function that computes the non-bonded interactions (Lennard Jones and electrostatic).
- (ii) *nbond* – The function that computes the pair list of atoms for the non-bonded interactions.

The above two routines (electrostatic and van der Waals interactions) are the only part of the MD calculations that require dynamic load balancing and parallelize in a less than perfect manner. Calculations of bonds, angles, torsions, etc. with fixed static lists parallelize perfectly.

- (iii) *mult\_mat\_vec* – Multiply matrix by a vector (part of the conjugate gradient algorithm).
- (iv) *build\_matrix\_from\_idx* – The build-up of the matrix  $A$  in its compressed form using the existing pointer.
- (v) *mult\_vec\_vec* – Scalar product of a vector by a vector (for the conjugate gradient algorithm).

Note also that *build\_matrix\_from\_idx* seems not to parallelize well. Since the matrix size and index are fixed, the parallelization of building the matrix can be efficient. However, we have made a choice to minimize the communication during matrix–vector multiplications, and we are computing some matrix elements on more than one processor. The repeats in computing matrix elements lead to an increase in computational cost for *build\_matrix\_from\_idx* and at the same time provide an almost perfect parallelization for *mult\_mat\_vec*. The matrix is computed only once in a time step while the matrix–vector multiplication is performed as many times as required by SHAKE convergence. For convenience we also provide the total time spent in SHAKE routines. The parallelization of the computing component (as opposed to communication) is good. Similar quality of parallelization is obtained for the direct MD integration. The bottleneck is the communication overhead as is reflected by the total time and the speed up factor.

## 9. Discussions and summary

We have presented a parallel implementation of SHAKE, an algorithm for solving classical equations of motion with holonomic constraints. The essence of the SHAKE parallelization was the use of a *minimization algorithm* to determine Lagrange’s multipliers. The conjugate gradient algorithm converges rapidly and is relatively easy to parallelize. The algorithm is implemented in MOIL [8], a publicly available molecular dynamics suite of programs (<http://cbsu.tc.cornell.edu/software/moil>). Our algorithm is exact in the sense that a parallel and a serial version are guaranteed to give the same answer. It also provides good load-balancing between the processors, as we demonstrated numerically. A key element of the new algorithm

Table 2  
Time distribution of critical subroutines in 1000 steps of dynamics for the membrane system

| Function  | No. of processors |        |        |        |
|---|-------------------|--------|--------|--------|
|   | 1                 | 4      | 8      | 16     |
| cdie  | 982               | 983.48 | 984.03 | 982.15 |
| watwat  | 701.73            | 702.2  | 701.66 | 699.7  |
| nbond   | 195.35            | 195.07 | 195.33 | 194.95 |
| mult_mat_vec  | 103.65            | 96.62  | 92.06  | 88.39  |
| nbondm  | 84.62             | 85.1   | 85.78  | 87.28  |
| conjugate_grad  | 34.87             | 17.97  | 15.87  | 14.97  |
| mult_vec_vec  | 20.61             | 19.85  | 20     | 21.23  |
| Total run time in seconds (summation over all processors) | 2280              | 2740   | 2872   | 3265   |
| Speed up factor   | 1                 | 3.33   | 6.35   | 11.19  |

The times quoted (in seconds) are a sum over all processors. Most of the functions were explained in the legend of Table 1. Other functions are

- (i) *watwat* – Compute non-bonded interactions between water molecules.
- (ii) *nbondm* – Create a coarse grained neighbor list to be refined in *nbond* to an atom list.

is the use of the matrix formulation with conjugate gradient optimization of Lagrange's multipliers. The constraints' coupling matrix,  $A_{ij}$ , and its matrix–vector product with Lagrange's multipliers is distributed between the CPUs. In essence the costly component of the SHAKE calculation becomes the parallelization of a vector-sparse matrix product, which can be done efficiently regardless of the method to parallelize the molecular dynamics algorithm (atomic or spatial decomposition). While the parallel algorithm is presented in the context of atom decomposition in which a complete set of the coordinates of the atoms is kept on each processor and the calculations of the forces (and the constraints) are parallelized, it should be straightforward to implement the algorithm for spatial decomposition. As long as the coupling matrix is dominantly diagonal, and the constraints directly couple only a few nearby atoms, a good correspondence between spatial decomposition of atoms and spatial decomposition of constraints (and the coupling matrix) can be worked out. The atom decomposition approach is more appropriate for small system and long-range interactions. The spatial decomposition approach is more appropriate for large systems with emphasis on short-range interactions. Overall spatial decomposition (under the conditions stated above) parallelizes better than the atom decomposition. The reality is that smaller systems and long-range interactions are more difficult to parallelize and the gains for small systems are therefore modest. On the other hand small systems and a few CPUs are common and modest gains that are broadly used are significant. Our choice of atom decomposition reflects the authors' interests in systems of moderate sizes rather than difficulties with the spatial decomposition approach.

We comment that the highly sparse nature of the coupling matrix in a broad range of applications, and its distributed storage over different CPUs make the memory requirements essentially linear with the system size. The algorithm is also able to detect when communication is not required and the matrix consists of blocks. For example, simulations of water solutions, membranes, or docking of two proteins will exploit the block structure of the matrix to minimize communication. Hence, even though tailored algorithms can be used for water solution and other aggregates of small molecules (for which the constraint coupling matrix has a block structure), similar performance is obtained using the current matrix formulation directly. We have obtained comparable performance for the membrane simulation with the current general algorithm and a tailored algorithm for rigid water molecules. In fact, the present algorithm is a lot better than the SHAKE implementation using bond relaxation for water molecules. The superior performance of the matrix SHAKE is a result of the triangular configuration of water molecules that is known to converge slowly with bond relaxation [2]. Of course, highly tuned algorithms for a specific molecule will win by the

end of the day, however, the generality and the uniform performance proposed by the current formulation is convenient.

## Acknowledgments

This work was supported in part by an NIH Grant GM059796 to R.E. Stimulating discussions with, and the careful reading of the manuscript by Giovanni Ciccotti are gratefully acknowledged. We thank Stacey Shirk for her editorial assistance.

## Appendix A

An intriguing special case for the matrix  $A$  is constrained bonds and bond angles. This type of constraints is widely used in polymer simulations in physics and chemistry, and (of course) does not imply that the molecule is rigid. Physically, constraining the angles is similar in concept to bond constraints. Both correspond to high frequency stiff motions even though bonds are stiffer than angles. As an example we note that the ECEEP force field for protein folding, developed in the group of Harold Scheraga, requires that bonds and bond angles will be fixed [17], i.e., exactly the case introduced in this appendix. Another example of simulations in dihedral angle space is shown in the research by Nobuhiro Go [20]. The formulation below will allow these studies to be done in the more convenient Cartesian formulation. We constrain the angles using an extra bond, and a concrete example is outlined below. If  $r_{ij}$  and  $r_{ik}$  are the vectors along the direction of the bonds and the angle is between  $r_{ij}$  and  $r_{ik}$ , we enforce the constraints by three distances

$$\sigma_1(r_{ij}) = r_{ij}^2 - d_{ij}^2 \quad \sigma_2(r_{ik}) = r_{ik}^2 - d_{ik}^2 \quad \sigma_3(r_{jk}) = r_{jk}^2 - d_{jk}^2. \quad (\text{A.1})$$

The matrix element  $A_{ll'}$  will be non-zero if the constraints  $\sigma_l$  and  $\sigma_{l'}$  share an atom (the mass matrix is diagonal). For bond constraints we have for the  $i$ th element of the constraint-force vector  $[\nabla\sigma_l(r_{ij})]_i = 2(r_i - r_j)$ . Let the constraint  $l$  be associated with the  $(i,j)$  bond and the bond  $(i,k)$  associated with the constraint  $l'$ . The  $A_{ll'}$  matrix is therefore

$$A_{ll'} = \begin{cases} l = l' & 2\Delta t^2 (r_i - r_j)^2 \left( \frac{1}{m_i} + \frac{1}{m_j} \right) \\ l \neq l' \text{ atoms are shared} & 2\Delta t^2 (r_i - r_j)^i \frac{1}{m_i} (r_i - r_k) \\ l \neq l' \text{ atoms not shared} & 0 \end{cases}. \quad (\text{A.2})$$

Note that if the constraints are satisfied exactly in the previous step,  $A_{ll'}$  is independent of time and can be constructed based on ideal distances and angles. The ideal distances being  $d_{ij}$  and angles  $\theta_{ijk}$ . We have

$$A_{ll'} = \begin{cases} l = l' & 2\Delta t^2 (d_{ij})^2 \left( \frac{1}{m_i} + \frac{1}{m_j} \right) \\ l \neq l' \text{ atoms are shared} & 2\Delta t^2 d_{ij} \frac{1}{m_i} d_{ik} \cos(\theta_{jik}) \\ l \neq l' \text{ atoms not shared} & 0 \end{cases}. \quad (\text{A.3})$$

Using Eq. (A.3) the matrix  $A_{ll'}$  and its inverse can be computed only once, at the beginning of the calculations and used anytime thereafter. Of course, if the number of constraints is large, calculating the inverse in advance may be too costly. If we wish to constrain only the bonds (and not the angles) then the above matrix  $A_{ll'}$  is not a constant (the constant version may still be useful as an approximation since both the bonds and the angles are restrained with harmonic potential in most force fields).

## References

- [1] H.C. Andersen, Rattle – a velocity version of the SHAKE algorithm for molecular dynamics calculations, *Journal of Computational Physics* 52 (1) (1983) 24–34.
- [2] E. Barth, K. Kuczera et al., Algorithms for constrained molecular dynamics, *Journal of Computational Chemistry* 16 (10) (1995) 1192–1209.
- [3] D. Brown, J.H.R. Clarke et al., A domain decomposition parallel processing algorithm for molecular dynamics simulations of polymers, *Computer Physics Communications* 83 (1993) 1–13.
- [4] D. Brown, H. Minoux et al., A domain decomposition parallel processing algorithm for molecular dynamics simulations of systems of arbitrary connectivity, *Computer Physics Communications* 103 (2–3) (1997) 170–186.
- [5] T.W. Clark, J.A. McCammon, Parallelization of a molecular dynamics nonbonded force calculation for mimd architecture, *Computers & Chemistry* 14 (3) (1990) 219–224.
- [6] T.H. Cormen, C.E. Leiserson et al., *Introduction to Algorithms*, second ed., MIT Press, Cambridge, MA, 2001.
- [7] S.E. Debolt, P.A. Kollman, AmberCube MD, parallelization of ambers molecular dynamics module for distributed-memory hypercube computers, *Journal of Computational Chemistry* 14 (3) (1993) 312–329.
- [8] R. Elber, A. Roitberg et al., Moil – a program for simulations of macromolecules, *Computer Physics Communications* 91 (1–3) (1995) 159–189.
- [9] G.D. Hawkins, C.J. Cramer et al., Pairwise solute descreening of solute charges from a dielectric medium, *Chemical Physics Letters* 246 (1–2) (1995) 122–129.
- [10] L. Kale, R.D. Skeel et al., NAMD2: greater scalability for parallel molecular dynamics, *Journal of Computational Physics* 151 (1) (1999) 283–312.
- [11] J.E. Mertz, D.J. Tobias et al., Vector and parallel algorithms for the molecular dynamics simulation of macromolecules on shared memory computers, *Journal of Computational Physics* 12 (10) (1991) 1270–1277.
- [12] F. Mullerplathe, D. Brown, Multicolor algorithms in molecular simulations – vectorization and parallelization of internal forces and constraints, *Computer Physics Communications* 64 (1) (1991) 7–14.
- [13] H.L. Nguyen, H. Khanmohammadbaigi et al., A parallel molecular dynamics strategy, *Journal of Computational Chemistry* 6 (6) (1985) 634–646.
- [14] J. Nocedal, S.J. Wright, *Numerical Optimization*, Springer, New York, 1999.
- [15] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, *Journal of Computational Physics* 117 (1) (1995) 1–19.
- [16] S. Plimpton, B. Hendrickson, A new parallel method for molecular dynamics simulation of macromolecular systems, *Journal of Computational Chemistry* 17 (3) (1996) 326–337.
- [17] D.R. Ripoll, M.S. Pottle et al., Implementation of the ECEPP Algorithm, the Monte Carlo minimization method and the electrostatically driven Monte-Carlo method on the Kendall Square Research KSRI Computer, *Journal of Computational Chemistry* 16 (9) (1995) 1153–1163.
- [18] J.P. Ryckaert, G. Ciccotti et al., Numerical-integration of Cartesian equations of motion of a system with constraints – molecular-dynamics of *N*-alkanes, *Journal of Computational Physics* 23 (3) (1977) 327–341.
- [19] R.D. Skeel, Macromolecular dynamics on a shared memory multiprocessor, *Journal of Computational Chemistry* 12 (2) (1991) 175–179.
- [20] M. Tomimoto, A. Kitao et al., Normal mode analysis of a nucleic acid with flexible furanose rings in dihedral angle space, *Electronic Journal of Theoretical Chemistry* 1 (1996) 122–134.
- [21] V. Tsui, D.A. Case, Theory and applications of the generalized Born solvation model in macromolecular Simulations, *Biopolymers* 56 (4) (2000) 275–291.
- [22] C.F. van Loan, *Introduction to Scientific Computing*, Prentice-Hall, Upper Saddle River, NJ, 1997.
- [23] L. Verlet, Computer experiments of classical fluids. I. Thermodynamical properties of Lennard-Jones molecules, *Physical Review* 159 (1) (1967) 98.
- [24] R. Zhou, R.A. Friesner et al., New linear interaction method for binding affinity calculations using a continuum solvent model, *Journal of Physical Chemistry* 105 (2001) 10388–10397.