Revisiting and parallelizing SHAKE

Yael Weinbach[1], and Ron Elber[2,*]

[1]The Hebrew University, Department of Computer Science, Givat Ram, Jerusalem 91904, Israel

and

[2]Cornell University, Department of Computer Science, Ithaca, NY 14853, USA

* Corresponding author:

e-mail ron@cs.cornell.edu, phone 607-255-7416, fax 607-255-4428

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.

Keywords: Dynamics with constraints, Conjugate gradient, Lagrange multipliers

I. 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 the Newton's equation

$$M\ddot{X} = -\nabla U \tag{1}$$

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

$$\begin{aligned} &M\ddot{X} = -\nabla U - \sum_l \lambda_l \nabla \sigma_l \\ &\text{subject to the constraints} \quad \sigma_l(X) = 0 \quad \forall l \end{aligned} \tag{2}$$

Calculations of the forces can take more than 90 percent of the computation time (if run serially). The estimates are based on the MOIL program (Elber, Roitberg et al. 1995) (see section on Numerical Examples below), 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 (Ryckaert, Ciccotti et al. 1977) a larger time step, typically by a factor of 2 to 3, can be used. The gain in computational time comes with the (small) penalty of 5 to 10 percent 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 (Nguyen, Khanmohammadbaigi et al. 1985; Clark and J.A. 1990; Mertz, Tobias D.J. et al. 1991; Mullerplathe and Brown 1991; Skeel 1991; Debolt and Kollman 1993; Plimpton 1995; Plimpton and Hendrickson 1996; Brown, Minoux et al. 1997; Kale, Skeel et al. 1999). 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 (Ryckaert, Ciccotti et al. 1977)) 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 (equation (11)). Two constraints, l and l', are directly coupled if $A_{ll'} \neq 0$, and

coupled if an integer n exists such that $\left(A^n\right)_{ll'} \neq 0$. 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. Consider 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 (Hawkins, Cramer et al. 1995; Tsui and Case 2000; Zhou, Friesner et al. 2001)).

Examples for previous heuristic approaches are as follows: Debolt and Kollman introduced a pipeline procedure for relaxation of bond lists (Debolt and Kollman 1993), and Brown et al. (Brown, Clarke et al. 1993) 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 percent 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 (Ryckaert, Ciccotti et al. 1977) and further developed in the work of Barth et al. (Barth E., Kuczera et al. 1995)) 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. (Barth E., Kuczera et al. 1995). 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 the "Discussion and Summary" section.

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.

II. 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 (Ryckaert, Ciccotti et al. 1977). 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 the Lagrange's multipliers from the differential equation are used, the numerical errors in the constraints (using a finite integration step Δt) 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 (Verlet L. 1967).

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

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 (Andersen 1983). The Lagrange's multipliers that exactly satisfy the constraints are not known, (i.e. $\sigma_l \left(X_{n+1} \right) = 0 \quad \forall l$), 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 $X_{n+1}^{(0)}$.

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

We use equation (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 equation (4) are the Lagrange's multipliers $\lambda_{l'}$. We define the matrix $A_{ll'}^{(0)}$

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

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

$$\sigma_l\left(X_i^{(0)}\right)=\sum_{l'}A_{ll'}^{(0)}\lambda_{l'} \tag{6}$$

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, Equation (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}

$$X_{n+1}^{(1)}=X_{n+1}^{(0)}-\frac{\Delta t^2}{2}M^{-1}\sum_l\lambda_{l,n}^{(0)}\nabla\sigma_{l,n} \tag{7}$$

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

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

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

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

And in more general terms

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

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

III. 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, $\sigma_l = \left(r_i - r_j \right)^2 - d_{ij}$ 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 to 4 bonds, the matrix is highly sparse, suggesting that a greedy algorithm (Cormen, Leiserson et al. 2001) 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 programming techniques, as discussed in the original paper of SHAKE (Ryckaert, Ciccotti et al. 1977). 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 (Mertz, Tobias D.J. et al. 1991). Here we exploit further the matrix formulation. In particular we note that usually a reasonable guess for the Lagrange's multipliers is available (e.g. the Lagrange's multipliers from the previous step). It therefore makes sense to build on this guess and to determine the 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 that the approach we finally used (see below) for two reasons: first, we need to compute matrix multiplication (an ℓ^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 (Barth E., Kuczera et al. 1995). We note that the difference between $\nabla\sigma_l\big|_{X=X_i^{(0)}}$ and $\nabla\sigma_l\big|_{X=X_i}$ 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

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

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 to 20 percent, 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 $F(x) = 1/2\, x^t A x - x^t b$ 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 ℓ (ℓ is the number of constraints) we have $a^t A a > 0$. The matrix $A_{ll'}$ can be represented as a sum of scalar products y_l and $y_{l'}$ ($y_l = \frac{\Delta t}{\sqrt{2}} \cdot M^{-1/2} \cdot \nabla \sigma_l$). The matrix element is

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

In the **Appendix** 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.

IV. 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 (Nocedal and Wright

1999)). We define a quadratic target function (in $\lambda_{l'}$) with a minimum at the desired solution for the Lagrange's multipliers

$$F = \frac{1}{2}\lambda^t A\lambda - \lambda^t\sigma \tag{13}$$

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 λ; as we discussed in the previous section, it is also possible to use non-symmetric A using minimization of a quadratic function in λ. 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 purse 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, α_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 λ_0 is determined by a single parameter α: $\lambda_1 = \lambda_0 + \alpha p_1 = \lambda_0 - \alpha g_1$. The gradient of the function at the minimum along the search line must be orthogonal to g_1. We therefore have:

$g_1 \cdot \left[A\left(\lambda_0 + \alpha p_1\right) - \sigma \right] = 0 \rightarrow -\alpha g_1^t A g_1 + g_1^t g_1 = 0$ 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 $\left(k \neq 1\right)$ the conjugate gradient search direction is $p_k = -g_k + \beta_k \cdot p_{k-1}$. The parameter β_k is chosen to ensure that the new search direction $\left(p_k\right)$ is "conjugate" to the previous search direction $\left(p_{k-1}\right)$, 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 ℓ steps (ℓ is the number of Lagrange's multipliers). In practice the number of steps can be reduced significantly (Nocedal and Wright 1999). 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 β_k is determined by $\beta_k = \frac{g_k^t g_k}{g_{k-1}^t g_{k-1}^t}$, 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 $\left(\left(B^{-1}\right)^t AB^{-1}\right)\eta = \left(B^{-1}\right)^t b$, which depending on the properties of B can accelerate the convergence significantly. We have used the diagonal of A to define a diagonal matrix $B = \left(A^D\right)^{1/2} = \left[diagonal\left(A\right)\right]^{1/2}$ with a trivial inverse. This simple precondition squeezed another iteration of two on our behalf. We define $C \equiv B^t B$, ε - maximum allowed error, $k_{\max}$ maximum number of iterations, and outline the algorithm below

Input: λ_1, and A, σ, C

Initiate loop:
$$k = 0$$
$$g_1 = A\lambda_1 - \sigma$$
$$\eta_1 = C^{-1} g_1$$
$$p_1 = -g_1$$

For $\left(\left|g_k\right| \geq \varepsilon \quad k = k+1 \quad k < k_{\max} - 1\right)$

$$
\begin{aligned}
\alpha_k &= \frac{g_k^t \eta_k}{p_k^t A p_k} \\
\lambda_{k+1} &= \lambda_k + \alpha_k p_k \\
g_{k+1} &= g_k + \alpha_k A p_k \\
\eta_{k+1} &= C^{-1} g_{k+1} \\
\beta_{k+1} &= \frac{g_{k+1}^t \eta_{k+1}}{g_k^t \eta_k} \\
p_{k+1} &= -\eta_{k+1} + \beta_{k+1} p_k
\end{aligned}
\tag{13}
$$

end

The computational efforts of the conjugate gradient algorithm, preconditioned or not, are focused on the matrix vector multiplication Ap_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 (van Loan 1997)), 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.

IV. 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 ℓ elements of V are the diagonal elements of A and the off-diagonal elements follow, one row at a time. The first ℓ 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 $V = \begin{pmatrix} 1.2 & 7.9 & 9 & 8.2 & 4.1 & 4.1 \end{pmatrix}$ and $P = \begin{pmatrix} 5 & 6 & 6 & 6 & 2 & 1 \end{pmatrix}$.

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 1589x1589 (99.7% of the elements are zero).

V. 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 (equation 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 the Lagrange's multipliers vector and its 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.

VI. The velocity constraints

The velocity constraints are implemented as described in RATTLE (Andersen 1983). We consider the constraint $\sigma_l(X)$ and differentiate it with respect to time

$$\frac{d\sigma_l(X)}{dt} = \frac{d\sigma_l}{dX} \cdot V = \nabla\sigma_l \cdot V = 0 \qquad \forall l \tag{14}$$

Of course, we wish the above scalar product to be zero for the newly generated step X_{n+1}. Note that we consider equation (14) only after we computed the coordinates of the next step. We therefore have from equation (3)

$$
\begin{aligned}
V_{n+1} &= V_n + \frac{\Delta t}{2} M^{-1}\left(-\nabla U_n - \sum_{l'} \lambda_{nl'} \nabla \sigma_{nl'} - \nabla U_{n+1} - \sum_{l'} \lambda_{n+1l'} \nabla \sigma_{n+1l'} \right) \\
V_{n+1} &= \bar{V}_n + \frac{\Delta t}{2} M^{-1}\left(-\nabla U_{n+1} - \sum_{l'} \lambda_{n+1l'} \nabla \sigma_{n+1l'} \right) = V_{n+1}^{(0)} - \frac{\Delta t}{2} M^{-1}\left(\sum_{l'} \lambda_{n+1l'} \nabla \sigma_{n+1l'} \right) \\
\bar{V}_n &= V_n + \frac{\Delta t}{2} M^{-1}\left(-\nabla U_n - \sum_{l'} \lambda_{nl'} \nabla \sigma_{nl'} \right) \\
V_{n+1}^{(0)} &= \bar{V}_n + \frac{\Delta t}{2} M^{-1}\left(-\nabla U_{n+1} \right)
\end{aligned}
\qquad (15)
$$

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

$$
V_{n+1} \cdot \nabla \sigma_{n+1,l}\Big|_{X=X_{n+1}} = V_n^{(0)} \cdot \nabla \sigma_{n+1,l}\Big|_{X=X_{n+1}} - \frac{\Delta t}{2} \sum_{l'} \lambda_{n+1,l'} \nabla \sigma_{n+1,l'} \cdot M^{-1} \nabla \sigma_{n+1,l}\Big|_{X=X_{n+1}} = 0 \qquad (16)
$$

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 (equation (14)) are linear in the Lagrange's multipliers. The linear equation (16) is therefore exact and no Newton's-like iterations are required to determine the Lagrange's multipliers.

VII. 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 11Å cutoff distance for electrostatic and 8Å for Lennard Jones interactions. The time step was 1 femtoseconds, and the temperature was 300K. The non-bonded lists were updated every 10 steps.

The parallel SHAKE algorithm discussed above was implemented in the program MOIL (Elber, Roitberg et al. 1995). 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 (Barth E., Kuczera et al. 1995). 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 one percent.

VIII. 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 the Langrange's multipliers. The conjugate gradient algorithm converges rapidly and is relatively easy to parallelize. The algorithm is implemented in MOIL (Elber, Roitberg et al. 1995), 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 loading balancing between the processors, as we demonstrated numerically. A key

element of the new algorithm is the use of the matrix formulation with conjugate gradient optimization of the Lagrange's multipliers. The constraints' coupling matrix, $A_{ll'}$, and its matrix-vector product with the Lagrange's multipliers is distributed between the CPU-s. 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 (Barth E., Kuczera et al. 1995). 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.

XI. Acknowledgements

This work was supported in part by an NIH grant GM059796 to RE. 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

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 (Ripoll, Pottle et al. 1995), 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 (Tomimoto, Kitao et al. 1996). 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

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

The matrix element $A_{ll'}$ will be non-zero if the constraints σ_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 $\left[\nabla\sigma_l\left(r_{ij}\right)\right]_i = 2\left(r_i - r_j\right)$. 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

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

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 θ_{ijk}. We have

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

Using equation (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

Andersen, H. C. (1983). "Rattle -- A velocity version of the SHAKE algorithm for molecular dynamics calculations." Journal of Computational Physics **52**(1): 24-34.

Barth E., K. Kuczera, et al. (1995). "Algorithms for constrained molecular dynamics." J Comput Chem **16**(10): 1192-1209.

Brown, D., J. H. R. Clarke, et al. (1993). "A domain decomposition parallel processing algorithm for molecular dynamics simulations of polymers." Computer Physics Communications **83**: 1-13.

Brown, D., H. Minoux, et al. (1997). "A domain decomposition parallel processing algorithm for molecular dynamics simulations of systems of arbitrary connectivity." Computer Physics Communications **103**(2-3): 170-186.

Clark, T. W. and M. J.A. (1990). "Parallelization of a molecular dynamics nonbonded force calculation for mimd architecture." Computers & Chemistry **14**(3): 219-224.

Cormen, T. H., C. E. Leiserson, et al. (2001). Introduction to algorithms, second edition. Cambridge, MA, MIT press.

Debolt, S. E. and P. A. Kollman (1993). "AmberCube MD, Paprallelization of Ambers Molecular DYnamics Module for Distributed-Memory Hypercube Computers." Journal of Computational Chemistry **14**(3): 312-329.

Elber, R., A. Roitberg, et al. (1995). "Moil - a Program for Simulations of Macromolecules." Computer Physics Communications **91**(1-3): 159-189.

Hawkins, G. D., C. J. Cramer, et al. (1995). "Pairwise Solute Descreening of Solute Charges from a Dielectric Medium." Chemical Physics Letters **246**(1-2): 122-129.

Kale, L., R. D. Skeel, et al. (1999). "NAMD2: Greater scalability for parallel molecular dynamics." Journal of Computational Physics **151**(1): 283-312.

Mertz, J. E., Tobias D.J., et al. (1991). "Vector and parallel algorithms for the molecular dynamics simulation of macromoleucles on shared memory computers." J Comput Chem **12**(10): 1270-1277.

Mullerplathe, F. and D. Brown (1991). "Multicolor algorithms in molecular simulations - Vectorization and parallelization of internal forces and constraints." Computer Physics Communications **64**(1): 7-14.

Nguyen, H. L., H. Khanmohammadbaigi, et al. (1985). "A parallel molecular dynamics strategy." J Comput Chem **6**(6): 634-646.

Nocedal, J. and S. J. Wright (1999). Numerical optimization. New York, Springer-Verlag.

Plimpton, S. (1995). "Fast parallel algorithms for short-range molecular dynamics." Journal of Computational Physics **117**(1): 1-19.

Plimpton, S. and B. Hendrickson (1996). "A new parallel method for molecular dynamics simulation of macromolecular systems." J Comput Chem **17**(3): 326-337.

Ripoll, D., R.,, M. S. Pottle, et al. (1995). "Implementation of the ECEPP Algorithm, the Monte Carlo Minimization Method and the Electrostatically Driven Monte-Carlo Method on the Kendall Square Research KSR1 Computer." Journal of Computational Chemistry **16**(9): 1153-1163.

Ryckaert, J. P., G. Ciccotti, et al. (1977). "Numerical-Integration of Cartesian Equations of Motion of a System with Constraints - Molecular-Dynamics of N-Alkanes." Journal of Computational Physics **23**(3): 327-341.

Skeel, R. D. (1991). "Macromolecular dynamics on a shared memory multiprocessor." J Comput Chem **12**(2): 175-179.

Tomimoto, M., A. Kitao, et al. (1996). "Normal mode analysis of a nucelic acid with flexible furanose rings in dihedral angle space." Electronic Journal of Theoretical Chemistry **1**: 122-134.

Tsui, V. and D. A. Case (2000). "Theory and applications of the generalized Born solvation model in macromolecular Simulations." Biopolymers **56**(4): 275-291.

van Loan, C. F. (1997). Introduction to scientific computing. Upper Saddle RIver, New Jersey, Prentice Hall.

Verlet L. (1967). "Computer experiments of classical fluids. I. Thermodynamical properties of Lennard-Jones molecules." Physical Review **159**(1): 98.

Zhou, R., R. A. Friesner, et al. (2001). "New linear interaction method for binding affinity calculations using a continuum solvent model." Journal of Physical Chemistry **105**: 10388-10397.

Table 1:

Time distribution of critical subroutines in 2000 steps of dynamics for the myoglobin system. The time step in all simulations was 1 femtosecond. 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 - 9,630 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.

No. of processors Function	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

Table 2

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

No. of processors Functions	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