Highly Efficient and Exact Method for Parallelization of Grid-Based Algorithms and its Implementation in DelPhi

Chuan Li,[a] Lin Li,[a] Jie Zhang,[a,b] and Emil Alexov*[a]

The Gauss–Seidel (GS) method is a standard iterative numerical method widely used to solve a system of equations and, in general, is more efficient comparing to other iterative methods, such as the Jacobi method. However, standard implementation of the GS method restricts its utilization in parallel computing due to its requirement of using updated neighboring values (i.e., in current iteration) as soon as they are available. Here, we report an efficient and exact (not requiring assumptions) method to parallelize iterations and to reduce the computational time as a linear/nearly linear function of the number of processes or computing units. In contrast to other existing solutions, our method does not require any assumptions and is equally applicable for solving linear and nonlinear equations. This approach is implemented in the DelPhi program, which is a finite difference Poisson–Boltzmann equation solver to model electrostatics in molecular biology. This development makes the iterative procedure on obtaining the electrostatic potential distribution in the parallelized DelPhi several folds faster than that in the serial code. Further, we demonstrate the advantages of the new parallelized DelPhi by computing the electrostatic potential and the corresponding energies of large supramolecular structures. © 2012 Wiley Periodicals, Inc.

DOI: 10.1002/jcc.23033

Introduction

The ability to calculate electrostatic forces and energies is critical in modeling biological molecules and nano-objects immersed in water and salt phase or another medium due to the fact that biological macromolecules are comprised of charged atoms. Their interactions and interactions with water and salt contribute to the structure, function, and interactions of biomolecules. At the same time, modeling the electrostatic potential of biological macromolecules is not trivial, and in a continuum case, requires solving the Poisson–Boltzmann equation (PBE)[1]

$$\nabla \bullet [\varepsilon(x)\nabla\phi(x)] - \kappa(x)^2 \sinh[\phi(x)] = -4\pi\rho(x), \qquad (1)$$

which is a second-order nonlinear elliptic partial differential equation discussed extensively in Ref. [2]. Here, $\phi(x)$ is the electrostatic potential, $\varepsilon(x)$ is the spatial dielectric function, $\kappa(x)$ is a modified Debye–Huckel parameter, and $\rho(x)$ is the charge distribution function.

The PBE does not have analytical solutions for irregularly shaped objects, and because of that, the solution must be obtained numerically. Numerous PBE solvers (PBES) have been designed and developed independently to use various mathematical methods to solve the PBE numerically. A short list includes AMBER,[3–6] CHARMM,[7] ZAP,[8] MEAD,[9] UHBD,[10] AFMPB,[11] matched interface and boundary-based poisson boltzmann equation software package (MIBPB),[12,13] ACG-based PBES,[14] Jaguar,[15] APBS, [16,17] and DelPhi.[1,18] Among these PBES, three popular implementations deserve specific attention in the light of current work. The APBS is a popular multigrid finite difference and adaptive finite element PBES developed by Dr. N. Baker and his colleagues and is aimed at providing force estimates and modeling large biomolecules and assemblages and p*K*a calculations. Another popular poisson boltzmann (PB) solver is MIBPB,[12,13,19–22] developed by Wei and coworkers which uses interface technique to assure potential and flux continuity at the interface biomolecule and solvent. Because of this and Krylov subspace technique[12,19–22] implementation, the MIBPB was demonstrated to be very robust PBES achieving second-order convergence for solving linear PBE.[13] The third popular solver is DelPhi, developed in Honig and coworkers laboratory,[1,18] which adopts the Gauss–Seidel (GS) method, combined with the successive over-relaxation (SOR) method which estimates the best relaxation parameter at run time [23] to solve both linear and nonlinear PBES. DelPhi has many unique features, such as abilities of modeling geometric objects (spheres, parallelepipeds, cones, and cylinders) and assigning multiple dielectric regions and charge distributions, and capabilities of allowing users to specify different types of salts and boundary conditions, as well as various output maps.

However, existing methods implemented in serial PBES are only suitable for electrostatic calculations of relatively small biomolecular systems due to time constraints. Nowadays, problems arising from computational biology are complex and usually of nano scale resulting in systems consisting of large

[a] *C. Li, L. Li, J. Zhang, E. Alexov*
Computational Biophysics and Bioinformatics, Department of Physics and Astronomy, Kinard Laboratory Building, Clemson University, SC 29634
E-mail: ealexov@clemson.edu

[b] *J. Zhang*
Department of Computer Science, Clemson University, Clemson, South Carolina 29642

Contract/grant sponsor: NIGMS, NIH; Contract/grant number: R01 GM093937.

amounts of charged atoms and tens of thousands of, even millions, mesh points. The size and complexity of these problems make parallelization of current serial PBES highly desired to improve their performance to solve the problems in reasonable time. For example, APBS was parallelized by a "parallel focusing" method based on the (spatial) domain decomposition method, together with standard focusing techniques. The results of parallel solution of the PBE for supramolecular structures, such as microtubule and ribosome structures, are presented in Ref. [17]. It should be mentioned that the solution obtained by this parallel method may not be identical to that obtained by the serial calculation, because to perform calculations in parallel on subsets of global mesh, additional values at boundaries of subsets must be, for instance, interpolated from the solution on a much coarser mesh in the first place.

In contrast to the (spatial) domain decomposition approach implemented in APBS, this article reports a novel approach to parallelize the GS method and its application to create a parallelized DelPhi. The implementation was facilitated due to the techniques already implemented in the serial DelPhi, such as the "checkerboard" ordering (also known as the "red-black" ordering[24]) and contiguous memory mapping,[1] to fulfill the GS method, and latter, the SOR method. In addition, the parallelization was possible because of the message passing interface version 1.0 (MPI-1), which allows the high-performance message-passing operations available for the advance distributed-memory communication environment supplied with parallel computers/clusters. Later on, MPI-2 was released to include new features such as parallel I/O, dynamic process management, and remote memory operations.[25] With the aid of powerful MPI libraries, we developed an efficient and exact method to parallelize the serial DelPhi (from the algorithmic point of view) to achieve linear/nearly linear speedup of its performance without compromising the accuracy and without introducing any assumptions. Although the approach was implemented in DelPhi, the very same parallelization technique can be translated and used by other software to implement/parallelize GS/SOR methods and other grid-based algorithms.

This article is organized as follows: (a) the techniques of implementing GS/SOR methods reported in Ref. [1] are described in the next section. Then, (b) parallel technique using MPI-2 remote memory operations is reported, and (c) implementation results and performance analysis on two examples of large supramolecular structures are demonstrated.

Efficient Implementation Techniques of the GS Method

In this section, we briefly describe the numerical methods and the techniques implemented in the serial DelPhi code. More details can be found in Supporting Information and original papers.[1,18,26]

Consider a three-dimensional (3D) cubical domain Ω. We discretize Ω into L grids per side with uniform grid size h. The total number of grid points is $N = L \times L \times L$. Let $K_0(x_0, y_0, z_0)$ be an arbitrary grid point away from the boundary of Ω. Applying finite difference formulation yields an iteration equation for Eq. (1):

$$\phi_0 = \frac{\sum_{i=1}^{6} \varepsilon_i \phi_i + 4\pi q_0/h}{\sum_{i=1}^{6} \varepsilon_i + (\kappa h)^2(\sinh(\phi_0))/\phi_0}, \tag{2}$$

where ϕ_0 and q_0 are the potential and charge assigned to K_0, ϕ_i, $i = 1, \ldots, 6$ are potentials at six nearest neighboring grids of K_0, and ε_i, $i = 1, \ldots, 6$ are the dielectric constants (taking value $\varepsilon_i = \varepsilon_{\text{out}}$ outside the protein and $\varepsilon_i = \varepsilon_{\text{in}}$ inside the protein), at midpoints of K_0 and its nearest neighbors (see Supporting Information for details). Equation (2) can be rewritten in matrix form as

$$\Phi = T\Phi + Q, \tag{3}$$

where T is the coefficient matrix and Φ and Q are column vectors.

Given appropriate boundary conditions at the edge of Ω and an initial guess for the potential at each grid point (usually zero for convenience), we may solve Eq. (3) iteratively using numerical methods such as Jacobi, GS, or SOR methods. In the case of serial calculations, the GS method is, in general, superior to the Jacobi method in the sense that it converges faster than the Jacobi method. The gain of convergence rate comes from the fact that the GS method uses latest updated potentials at neighboring points in current iteration, instead of values obtained in the previous iteration as in the Jacobi method. However, without special treatment, the requirement of using latest updated neighboring values makes the GS method less favorable to parallel computing due to the fact that calculations at one point cannot start before the completion of calculations at its neighbors. To efficiently parallelize the GS method, an implementation technique, called the "checkerboard" ordering,[1] which has been implemented in serial DelPhi, will be discussed in the following section.

The "checkerboard" ordering

Solving Eq. (3) iteratively requires construct mapping to convert potentials and charges at 3D grid points to column vectors Φ and Q. One common mapping, alternating index in x-direction first, followed by indices in y- and z-direction, is given by

$$w = x + L \times (y - 1) + L^2 \times (z - 1), \tag{4}$$

which maps the potential and charge at point $P(x,y,z)$ to $\Phi(x)$, and $Q(w)$, respectively.

The associated coefficient matrix T is determined by the order in which the grid points are mapped. However, it has been pointed out in Ref. [27] that this mapping does not affect the spectral radius of matrix T. That is, the convergence rate of the iteration method is independent of the mapping order, which allows us to reorder the components of Φ and Q, and reconstruct associated matrix T in desired fashion without losing the overall convergence rate of the iteration method.

Another important observation on the grid-to-vector mapping is that each grid point $P(x,y,z)$ can be assigned as odd or even by the sum of its grid coordinates, sum $= x + y + z$.[1]

We call P an even point if sum is even, and P an odd point if sum is odd. The six nearest neighbors of P are of opposite nature, that is, every even point is surrounded by odd points and vice versa because the sum of their coordinates only differ by one. As shown in Eq. (2), updating of the potential at any point only depends on the potentials at its six nearest neighbors, we see that even is updated by surrounding odds, and odd is updated by surrounding evens. Moreover, provided L an odd number, index w obtained by Eq. (4) is of the same even/odd nature as sum which leads to the following reorganization of Φ and Q simply by sum

$$\Phi = \begin{bmatrix} \Phi_{\text{even}} \\ \Phi_{\text{odd}} \end{bmatrix}, \quad Q = \begin{bmatrix} Q_{\text{even}} \\ Q_{\text{odd}} \end{bmatrix}, \tag{5}$$

where Φ_{even} and Q_{even} are potentials and charges at even points, and Φ_{odd} and Q_{odd} are those at odd points. The corresponding coefficient matrix T is then of the form

$$T = \begin{bmatrix} 0 & T_{\text{odd}} \\ T_{\text{even}} & 0 \end{bmatrix}, \tag{6}$$

such that T_{odd} is the submatrix which updates Φ_{even} with Φ_{odd}, and in turn, T_{even} updates Φ_{odd} with newly obtained Φ_{even}. Equation (3) is thereby equivalent to

$$\begin{cases} \Phi_{\text{even}} = T_{\text{odd}}\Phi_{\text{odd}} + Q_{\text{even}} \\ \Phi_{\text{odd}} = T_{\text{even}}\Phi_{\text{even}} + Q_{\text{odd}} \end{cases}. \tag{7}$$

Equation (7) allows the GS method to be implemented in Jacobi's fashion and makes it suitable for parallelization.

Contiguous memory mapping

Before we move to the next section introducing techniques on how to parallelize the GS scheme effectively, one more implementation technique, namely contiguous memory mapping, needs to be described.

It is noticed[1] that the performance of the "checkerboard" ordering implementing the GS method is slowed down using a logical operation ("IF" statement) in the most inner loop of the algorithm to separate the process of updating Φ into odd and even cycles. Considering a case such that Φ is composed of millions of points and the numerical algorithm requires hundreds of iterations to converge, the cost of this logical operation is unaffordable. Therefore, it was suggested in Ref. [1] that the best way to efficiently code the ordering is to map the odd and even points separately into two contiguous memory/arrays, that is, Φ_{odd} and Φ_{even}. This leads to a more complex coding of the algorithm but avoids branching the inner loop.[1]

Discussions of Parallelization Techniques and Parallel Algorithm

After implementing the techniques described earlier, parallelizing Eq. (7) of the GS method is conceptually straightforward: provided N_{cpu} processes or computing units (CPUs) at our disposal, we divide Φ_{odd} and Φ_{even} evenly into N_{cpu} segments. Each pair of segments of Φ_{odd} and Φ_{even} is given to one CPU for updating. To reproduce values obtained from serial calculations without imposing additional boundary conditions, additional memory is allocated to synchronize values near both ends of the segments. Synchronization takes place right after the segments of Φ_{odd} and Φ_{even} are updated locally.

Implementing the above idea effectively requires network communication to be reduced. An efficient algorithm must minimize the ratio between the amount of data to be synchronized and the amount of data to be computed locally per CPU, that is, the CPU must spend more time computing than communicating.

Details of the parallelization

Notice Φ is of the length L^3 and therefore, segments of Φ_{odd} and Φ_{even} to be updated locally per CPU are of the same length $L^3/(2N_{\text{CPU}})$. Let $p_0(x_0, y_0, z_0)$ be an even point and the potential at p_0 is mapped to $\Phi_{\text{even}}(w_0)$ with $w_0 = \frac{x_0+L(y_0-1)+L^2(z_0-1)+1}{2}$, the potentials at six nearest neighbors of p_0 are then mapped to six entries of Φ_{odd} shown in the second column of Table 1. Similarly, when p_0 is odd and

Table 1. Potentials at six nearest neighbor points of p_0 in $\boldsymbol{\Phi}_{\text{odd}}$ and $\boldsymbol{\Phi}_{\text{even}}$.

Neighboring points of $p_0(x_0, y_0, z_0)$	Entries of Φ_{odd} when p_0 is even	Entries of Φ_{even} when p_0 is odd
$p_1(x_0 - 1, y_0, z_0)$	$\Phi_{\text{odd}}(w_0 - 1)$	$\Phi_{\text{even}}(w_0)$
$p_2(x_0 + 1, y_0, z_0)$	$\Phi_{\text{odd}}(w_0)$	$\Phi_{\text{even}}(w_0 + 1)$
$p_3(x_0, y_0 - 1, z_0)$	$\Phi_{\text{odd}}\left(w_0 - \frac{L+1}{2}\right)$	$\Phi_{\text{even}}\left(w_0 - \frac{L-1}{2}\right)$
$p_4(x_0, y_0 + 1, z_0)$	$\Phi_{\text{odd}}\left(w_0 + \frac{L-1}{2}\right)$	$\Phi_{\text{even}}\left(w_0 + \frac{L+1}{2}\right)$
$p_5(x_0, y_0, z_0 - 1)$	$\Phi_{\text{odd}}\left(w_0 - \frac{L^2+1}{2}\right)$	$\Phi_{\text{even}}\left(w_0 - \frac{L^2-1}{2}\right)$
$p_6(x_0, y_0, z_0 + 1)$	$\Phi_{\text{odd}}\left(w_0 + \frac{L^2-1}{2}\right)$	$\Phi_{\text{even}}\left(w_0 + \frac{L^2+1}{2}\right)$

the potential at p_0 is mapped to $\Phi_{\text{odd}}(w_0)$ with $w_0 = \frac{x_0+L(y_0-1)+L^2(z_0-1)}{2}$, the potentials at its six neighbors are shown in the third column of Table 1.

We can see from Table 1 that, on each CPU, at most $\frac{L^2+1}{2} + \frac{L^2-1}{2} = L^2$ elements near the ends of segments of Φ_{odd} and Φ_{even} are required to be synchronized in iterations. Therefore, the ratio of the number of elements to be exchanged and the number of elements to be calculated locally is

$$r = \frac{L^2/2}{L^3/(2N_{\text{CPU}})} = \frac{N_{\text{CPU}}}{L}. \tag{8}$$

Equation (8) provides an insight to the speedup, efficiency, and scalability of the parallel algorithm. For example, small r ($r \ll \mathbf{1}$, or equivalently, $N_{\text{CPU}} \ll L$) indicates that communication cost in the parallel computation contributes only a minor portion of the overhead, assuming cost of network communication is comparable to that of CPU floating point calculations.

Table 2. An algorithm for parallelizing iterations in the GS method using MPI-2 DRMA operations.

Step 1: The master CPU assigns 3D points as even and odd according to the sum of its coordinates. An arbitrary grid point $P(x,y,z)$ and its six nearest neighbors are shaded and shown in Figure 1a. The master CPU maintains Q_{even}, Q_{odd}, Φ_{even}, and Φ_{odd}. Relative positions of p in Φ_{even} and its neighbors in Φ_{odd} are demonstrated in Figure 1b. Q_{even}, Q_{odd}, Φ_{even}, and Φ_{odd} are then divided evenly and distributed to every slave CPU.
Step 2: Each slave CPU allocates contiguous memory for segments of Q_{even} and Q_{odd} of length $L^3/(2N_{\text{CPU}})$, and segments of Φ_{even} and Φ_{odd} of length $L^3/(2N_{\text{CPU}}) + L^2$. In Φ_{even} and Φ_{odd}, the first and last $L^2/2$ elements (unshaded regions in Fig. 1c) are for receiving updated potentials on the previous and next CPUs, and, in between (shaded regions in Fig. 1c), $L^3/(2N_{\text{CPU}})$ potentials are updated locally on this CPU. The colored regions in Figure 1c indicate where synchronization takes place.
Step 3: The first $(N_{\text{CPU}}-1)$ slave CPUs create two windows, win1 and win2, of size L^2 at the right end of Φ_{even} and Φ_{odd} by MPI_WIN_CREATE (shown in Fig. 1d).
Step 4: Slave CPU(i) opens window win2 using MPI_WIN_POST. CPU(i+1) starts request of DRMA using MPI_WIN_START, copies $L^2/2$ elements in the shaded red region of Φ_{odd} to the unshaded red region of Φ_{odd} on CPU(i) by MPI_PUT, brings another $L^2/2$ elements in the shaded blue region of Φ_{odd} on CPU(i) back to the unshaded blue region of Φ_{odd} on CPU(i+1) by MPI_READ, and completes the request by MPI_WIN_COMPLETE. The whole access epoch is completed after CPU(i) calls MPI_WIN_WAIT, as shown in Figure 1d.
Step 5: Each slave CPU updates the elements of Φ_{even} in the shaded region one by one using Eq. (2), or more efficiency method described in Ref. [1].
Step 6: Follow the same procedures in Steps 4–5 to update Φ_{odd} using updated Φ_{even}.
Step 7: Repeat Steps 4–6 until predefined tolerance or the maximal number of iterations is achieved. The computed results are sent back to the master CPU for reassembling.

In such cases, linear speedup of the parallel computing is very likely to be achieved.

Many other factors may affect the performance of the parallelized code. In particular, appropriate MPI communication operations must be chosen carefully to avoid unwanted delays because of queue in synchronization. Synchronization across all processors can be achieved by either blocking or nonblocking operations provided by MPI. Blocking operations require the sender to wait for receiving the confirmation from the receiver before the sender can process to the next operation. In our case, one processor needs to exchange boundary values with both neighboring processors on its left and right sides. If blocking operations are chosen to use, in the worst scenario, processor 2 needs to talk to processor 1 and wait for response from processor 1 before it can talk to processor 3, and so on. In this case, a communication queue is created and the last processor communicates the last. It is obviously not efficient. Therefore, nonblocking operations are more favorable for this application. Moreover, because the computer cluster is equipped with Myrinet, on which one-sided operations have potentials to perform better than two-sided operations, one-sided direct memory access operations were used in this method.

MPI-2 library provides two communication models: two-sided communication based on blocking/nonblocking send and receive operations and one-sided communication allowing direct remote memory access (DRMA) of a remote process.[28] Two-sided communication requires actions on both sides of sender and receiver. In contrast, one-sided communication specifies communication parameters only on the 'requester' side (called the origin process) and leaves the 'host' (called the target process) alone without interrupting its ongoing work during communication. One-sided communication requires additional operations to create an area of memory (call a 'window') in the target process for the origin process to access before communication takes place.

One-sided communication fits in our requirements very well. It is more convenient to use and has the potential to perform better on the networks like InfiniBand and Myrinet, where one-sided communication is supported natively.[29] One-side communication requires explicit synchronization to ensure the completion of communication. Three synchronization mechanisms are provided in MPI-2:[30] the fence synchronization, the lock/unlock synchronization, and the post-start-complete-wait synchronization. Among these three synchronization mechanisms, the scope of the post-start-complete-wait synchronization can be restricted to only a pair of communicating, which makes it the best candidate in our scenario: synchronizations between two successive processes take place at almost the same time, and moreover, provided the problem size is fixed [see Eq. (8)], the amount of data to be exchanged between two processes is of the same no matter how many processes are allocated.

Parallel algorithm

In the light of above results and discussions, we followed the Master-Slaves paradigm and developed an efficient and exact algorithm to parallelize the iterations in the GS method using MPI-2 one-sided DRMA operations. The algorithm is described in Table 2 and graphically shown in Figure 1.

Implementation Results and Conclusions

Analyzing the performance of parallel programs requires background in parallel computing. For readers who are not familiar with parallel computing, it is suggested to refer to the supporting information for some commonly used quantities, such as speedup and efficiency, as well as some important theoretical results, for performance analysis of parallel algorithms.

The numerical experiments reported in this section were done with parallelized DelPhi (version 5.1, written in FORTRAN95) and performed on two types of computer nodes of the Palmetto cluster at Clemson University:[31] (I) Computer node 1112-1541 of Sun X6250 with Intel Xeon L5420 at 2.5 GHz x2 processors, 8 cores, 6 MB L2 cache, and 32 GB memory; (II) Computer node 1553-1622 of HP DL 165 G7 with AMD Opteron 6172 at 2.1 GHz x2 processors, 24 cores, 12 MB L2 cache, and 48 GB memory. Myrinet network (10 GB) is equipped on the Palmetto cluster. All experiments, except those of serial nonlinear implementations which require more than 30 GB memory, were performed on the first set of nodes for consistency. Each identical

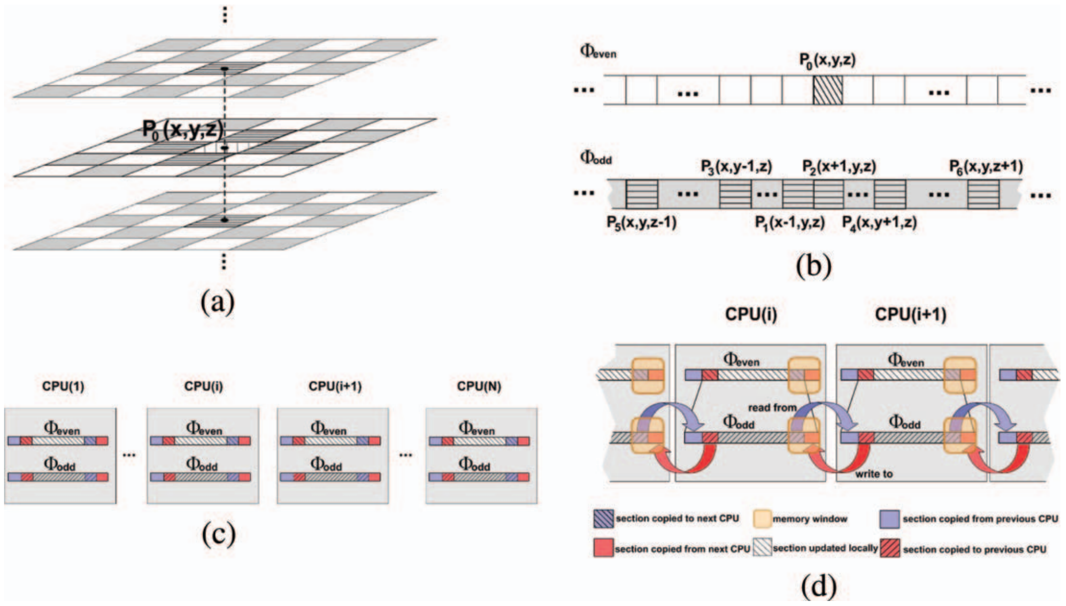
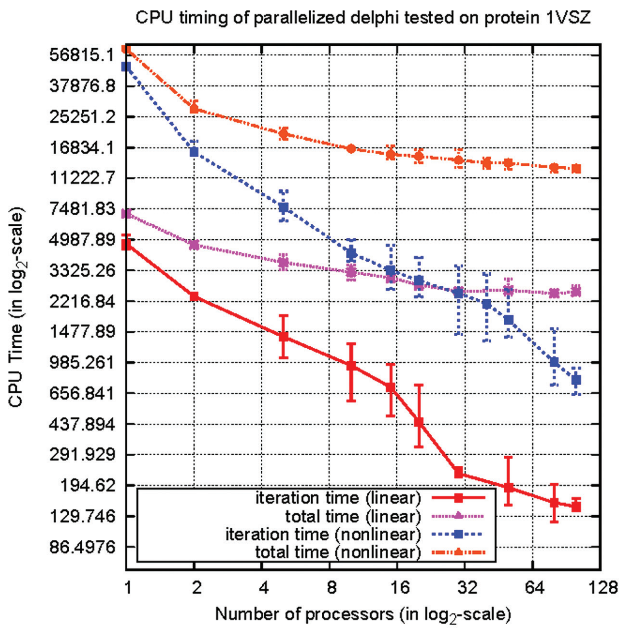
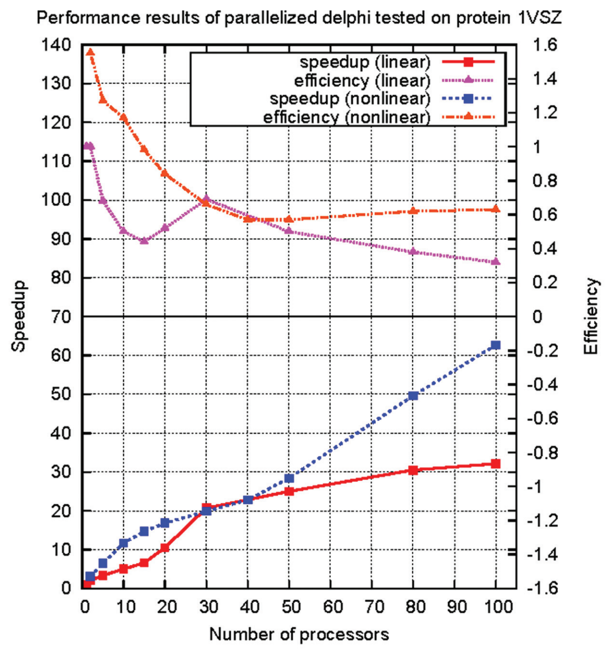


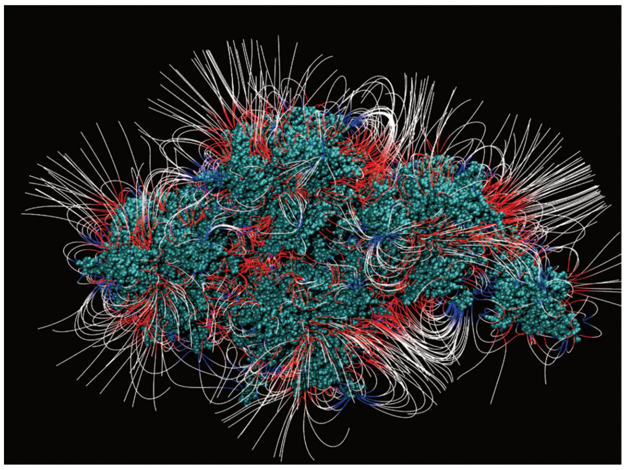
Figure 1. Graphical demonstration of an algorithm for parallelizing the iterations in the GS/SOR method uisng MPI-2 DRMA operations. a) The 'checker-board' ordering. b) Contiguous memory mapping. c) Distribution of Φ_{even} and Φ_{odd} to multiple CPUs. d) DRMA to the previous CPU. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]



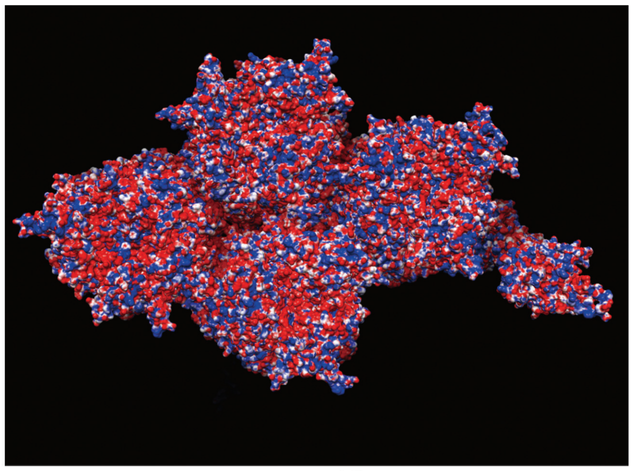
(a)



(b)



(c)



(d)

Figure 2. Performance results and electrostatic properties of 1VSZ. a) Execution (purple) and iteration (red) time for solving the linear PBE, compared to execution (orange) and iteration (blue) time for solving the nonlinear PBE. b) Speedup (red) and efficiency (purple) achieved by solving the linear PBE, compared to speedup (blue) and efficiency (orange) obtained by solving the nonlinear PBE. c) Resulting electrostatic field. d) Resulting electrostatic potential.

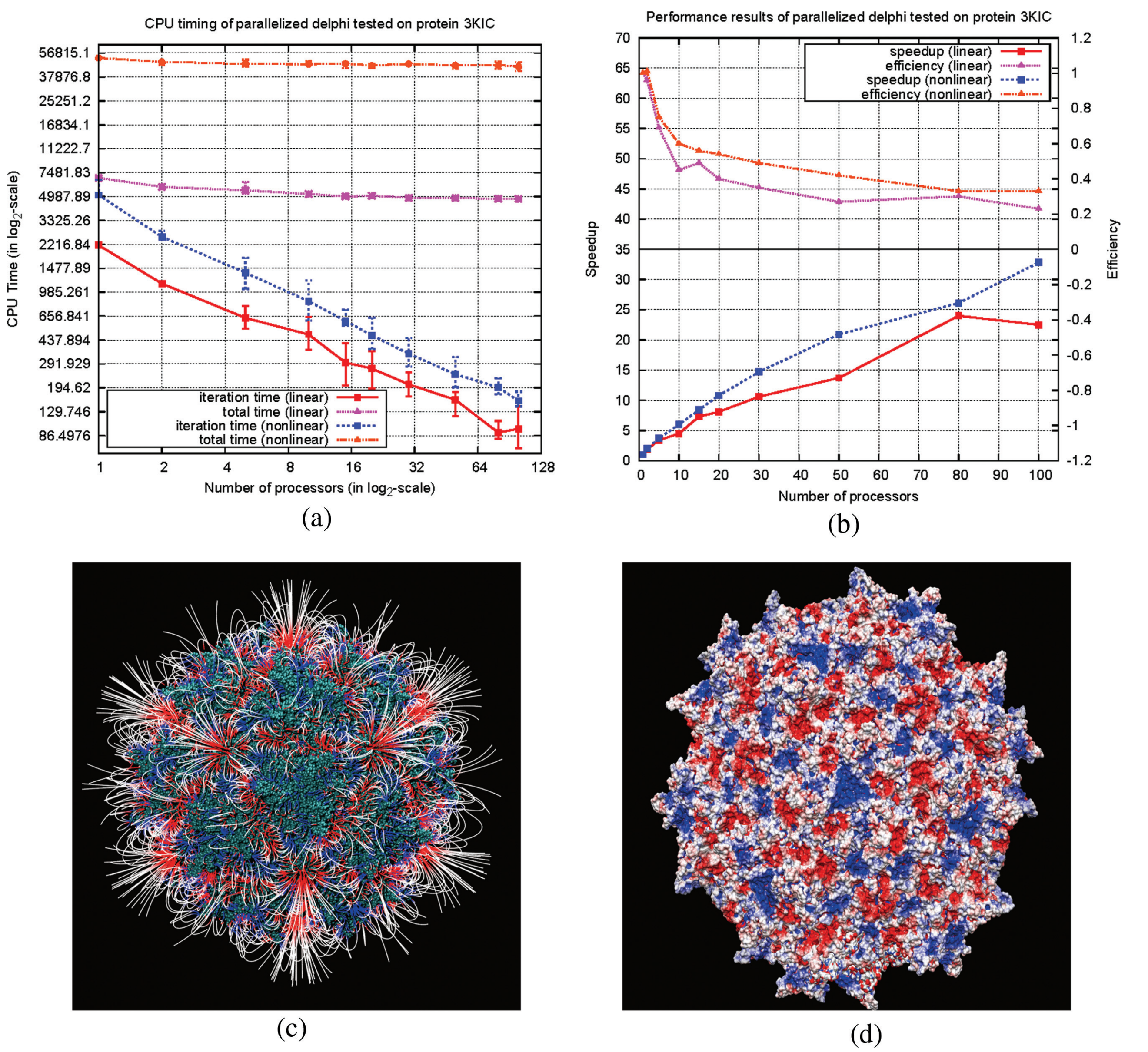


Figure 3. Performance results and electrostatic properties of 3KIC. a) Execution (purple) and iteration (red) time for solving the linear PBE, compared to execution (orange) and iteration (blue) time for solving the nonlinear PBE. b) Speedup (red) and efficiency (purple) achieved by solving the linear PBE, compared to speedup (blue) and efficiency (orange) obtained by solving the nonlinear PBE. c) Resulting electrostatic field. d) Resulting electrostatic potential.

experiment was repeated five times and the average is reported here to reduce random fluctuations caused by system workload and network traffic in real time.

All calculations used the same Amber force field. Scale = 2.0 and 70% filling of the box domain were set in the parameter file resulting in the dimensions of the box domain $\approx 407 \times 407 \times 407$ Å^3 and $815 \times 815 \times 815$ mesh points in total.

The first series of experiments, requiring solving linear and nonlinear PB equations, were performed on a fraction of the protein of human adenovirus 1VSZ downloaded from the Protein Data Bank[32] and protonated by TINKER.[33] The CPU time achieved by solving the linear and nonlinear PBE as a function of increasing number of processors is shown in Figure 2a with vertical bars indicating variations of five runs. To compare their performance, log-scale plots of speedup and efficiency are shown in Figure 2b. The resulting potential and electrostatic field are plotted by visual molecular dynamics software package[34] and demonstrated in Figures 2c and 2d.

The next series of experiments were performed on the protein of adeno-associated virus 3KIC, which has significantly more atoms ($\approx$484,500 atoms in a pdb file of size 25.4 MB) than those of 1VSZ ($\approx$180,574 atoms in a pdb file of size 9.5 MB). The results are presented in Figures 3a–3d.

It should be emphasized that the reported parallelization and its implementation in DelPhi are exact. No approximations were made. This is demonstrated by the fact that the calculated potentials and energies are identical for serial and parallelized DelPhi (see Supporting Information). The importance of achieving exact solution stems from the fact that in many biologically relevant cases, the potential and energy differences are of order of several kT/e or kT units or even less. Any assumption may induce an error larger than that, especially if computing large size systems, and thus to obscure the outcome.

It was shown that the parallelization drastically improves the speed of calculations, especially in case of solving nonlinear PBE. The case of 1VSZ was purposely included in the testing, although the structure represents only part of the capsid, simply to illustrate a case of highly charged entity with very irregular (different from sphere) shape. This particular case, at the limits of our testing, using 100 CPUs resulted in speed up of 63 for solving nonlinear PBE. This illustrates that problems requiring heavy computations will benefit from parallelization substantially. Equation (8) provides an efficient formula to estimate the conditions at which the parallel algorithm will be outperforming the serial one. Obviously, the cases involving large systems made of protein complexes will be the primary choice of investigation with the parallel DelPhi. In another words, the speedup of the parallel algorithm will depend on the ratio of the CPU time and communication time, as indicated by Eq. (8). With decrease of the size of the system, as small biomolecules with small mesh, the CPU time will decrease, making the coefficient "*r*" in Eq. (8) larger and reducing the efficiency of the algorithm. Because of that calculations involving small biomolecules are not expected to take advantage of this approach.

Analysis of Figures 2b and 3b reveals another important aspect of parallelization in case of solving nonlinear PB. The speedup is almost linear when running on a small number of processors. It keeps increasing with the increase of processors (up to 100 processors in our test) and even shows potential to increase further because its curve has not reached its peak and flattened out, as pointed out in the Amdahl's law (introduced in Supporting Information). The best result we obtained is a speedup of 63 when running on 100 processors to solve the nonlinear PBE for 1VSZ. At the same time, the efficiency decreases slowly when the number of used processors increases for solving both linear and nonlinear PBE (Figs. 2b and 3b). This observation confirms our previous discussion and reflects the ratio outlined in Eq. (8). However, one can see that the new parallel method maintains better efficiency when solving nonlinear PBE, because more computations are involved comparing with solving linear PBE. Moreover, Figure 2b shows that "super-linear" speedup is achieved when solving nonlinear PBE for 1VSZ on less than 15 processors (see the beginning of the graph when only a few processors are involved in the calculations). It is another indication of the high efficiency of the algorithm.

Acknowledgments

The authors thank Shawn Witham for reading the manuscript before submission.

Keywords: electrostatics · DelPhi · Poisson–Boltzmann equation · Gauss–Seidel iteration · parallel computing

How to cite this article: C. Li, L. Li, J. Zhang, E. Alexov, *J. Comput. Chem.* **2012,** *33,* 1960–1966. DOI: 10.1002/jcc.23033

Additional Supporting Information may be found in the online version of this article.

[1] A. Nicholls, B. Honig, *J. Comput. Chem.* **1991**, *12,* 435.
[2] M. K. Gilson, A. Rashin, R. Fine, B. Honig, *J. Mol. Biol.* **1985**, *184,* 503.
[3] D. A. Case, T. E. Cheatham, III, T. Darden, H. Gohlke, R. Luo, K. M. Merz, Jr., A. Onufriev, C. Simmerling, B. Wang, R. J. Woods, *J. Comput. Chem.* **2005**, *26,* 1668.
[4] R. Luo, L. David, M. K. Gilson, *J. Comput. Chem.* **2002**, *23,* 1244.
[5] M. J. Hsieh, R. Luo, *Proteins* **2004**, *56,* 475.
[6] C. Tan, L. Yang, R. Luo, *J. Phys. Chem. B* **2006**, *110,* 18680.
[7] B. R. Brooks, C. Brooks, III, A. Mackerell, Jr., L. Nilsson, R. Petrella, B. Roux, Y. Won, G. Archontis, C. Bartels, S. Boresch, *J. Comput. Chem.* **2009**, *30,* 1545.
[8] J. A. Grant, B. T. Pickup, A. Nicholls, *J. Comput. Chem.* **2001**, *22,* 608.
[9] D. Bashford, In Lecture Notes in Computer Science; Y. Ishikawa, R. Oldehoeft, J. Reynders, M. Tholburn, Eds.; Springer Berlin: Heidelberg, **1997**; pp. 233–240.
[10] M. E. Davis, J. A. McCammon, *J. Comput. Chem.* **1989**, *10,* 386.
[11] B. Lu, X. Cheng, J. Huang, J. A. McCammon, *J. Chem. Theory Comput.* **2009**, *5,* 1692.
[12] S. Yu, G. Wei, *J. Comput. Phys.* **2007**, *227,* 602.
[13] D. Chen, Z. Chen, C. Chen, W. Geng, G. W. Wei, *J. Comput. Chem.* **2011**, *32,* 756.
[14] A. H. Boschitsch, M. O. Fenley, H. X. Zhou, *J. Phys. Chem. B* **2002**, *106,* 2741.
[15] C. M. Cortis, R. A. Friesner, *J. Comput. Chem.* **1997**, *18,* 1591.
[16] M. Holst, N. Baker, F. Wang, *J. Comput. Chem.* **2000**, *21,* 1319.
[17] N. A. Baker, D. Sept, S. Joseph, M. J. Holst, J. A. McCammon, *Proc. Natl. Acad. Sci. USA* **2001**, *98,* 10037.
[18] I. Klapper, R. Hagstrom, R. Fine, K. Sharp, B. Honig, *Proteins* **1986**, *1,* 47.
[19] S. Zhao, G. Wei, *J. Comput. Phys.* **2004**, *200,* 60.
[20] Y. Zhou, S. Zhao, M. Feig, G. Wei, *J. Comput. Phys.* **2006**, *213,* 1.
[21] Y. Zhou, G. Wei, *J. Comput. Phys.* **2006**, *219,* 228.
[22] S. Yu, Y. Zhou, G. Wei, *J. Comput. Phys.* **2007**, *224,* 729.
[23] S. Sridharan, A. Nicholls, B. Honig, *Biophys. J.* **1992**, *61,* A174.
[24] J. W. Demmel, Using MPI-2: Advanced Features of the Message Passing Interface. xi+ 419 pp. Demmel: William Gropp, Ewing Lusk, Anthony Skjellum, MIT Press, ISBN 0-262-57133-1, **1999**.
[25] W. Gropp, E. Lusk, R. Thakur, Applied Numerical Linear Algebra. By James W. Demmel. SIAM, Philadelphia, PA, 1997. Xi+419 pp., softcover: ISBN 0-89871-389-7. MIT Press: Cambridge, MA, **1999**.
[26] B. Jayaram, K. A. Sharp, B. Honig, *Biopolymers* **1989**, *28,* 975.
[27] R. Bulirsch, J. Stoer, Introduction to Numerical Analysis; Springer-Verlag: New York, **1980**.
[28] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, M. Snir, In Lecture Notes in Computer Science; L. Bougé, P. Fraigniaud, A. Mignotte, Y. Robert, Eds.; Springer Berlin: Heidelberg, **1996**; pp. 128–135.
[29] R. Thakur, W. Gropp, B. Toonen, *Int. J. High Perform. Comput. Appl.* **2005**, *19,* 119.
[30] B. Barrett, G. Shipman, A. Lumsdaine, Recent Advances in Parallel Virtual Machine and Message Passing Interface, Analysis of Implementation Options for MPI-2 one-sided, Vol. 4757/2007, pp. 242–250. Springer Berlin / Heidelberg, **2007**, DOI: 10.1007/978-3-540-75416-9_35.
[31] C. Galen, Palmetto Cluster User Guide, available at: http://desktop2petascale.org/resources/159. Accessed on May 22, 2012.
[32] F. C. Bernstein, T. F. Koetzle, G. J. B. Williams, E. F. Meyer, Jr., M. D. Brice, J. R. Rodgers, O. Kennard, T. Shimanouchi, M. Tasumi, *Eur. J. Biochem.* **1977**, *80,* 319.
[33] J. Ponder, F. Richards, *J. Comput. Chem.* **1987**, *8,* 1016.
[34] W. Humphrey, A. Dalke, K. Schulten, *J. Mol. Graph.* **1996**, *14,* 33.

Received: 17 February 2012
Revised: 20 April 2012
Accepted: 11 May 2012
Published online on 4 June 2012