Journal of Chemical Theory and Computation

The Use of Processor Groups in Molecular Dynamics Simulations to Sample Free-Energy States

Bruce Palmer,* Shawn Kathmann, Manojkumar Krishnan, Vinod Tipparaju, and Jarek Nieplocha

Pacific Northwest National Laboratory, Box 999, Richland, Washington 99352

Received August 9, 2006

Abstract: Molecular dynamics calculations composed of many independent simulations are frequently encountered in free-energy calculations, as well as many other simulation approaches. In principle, the availability of a large number of independent tasks should make possible the development of highly scalable parallel code that executes these tasks concurrently. This paper discusses the use of processor groups to write simulation codes of this type and describes results for a code that evaluates the volume dependence of the Helmholtz free energy for clusters of an immiscible fluid in a solvent. The results show that very high levels of scalability can be achieved using processor groups with corresponding reductions in the time to completion. The main limitation to scaling appears to be a load imbalance due to variations in the execution times of the individual tasks.

Introduction

Molecular dynamics (MD) simulations are becoming an increasingly important tool in understanding the properties and behavior of complex chemical and biochemical systems. Biochemists routinely use simulations to help understand problems such as protein−ligand interactions, the binding of drugs to active sites, and the role of conformation changes in enzymatic activity.[1] Simulations are also being used to understand many problems in materials science and chemistry, particularly with regard to reactions in aqueous phases.[2] The higher speeds and decreasing cost of processors have also contributed to the widespread use of molecular dynamics. Simulations that previously took days on a high-end supercomputer can now be performed on a relatively inexpensive workstation. Furthermore, the availability of clusters and other parallel architectures has continued to push the bounds on the size of the simulation that can be performed, to the point where the main limitation is, in many cases, not the size of the system that can be simulated but rather the time interval over which the simulation can be performed. For many problems, the goal is to simulate a fixed-size system for longer periods of time or to obtain better ensemble averages rather than increasing the size of the system.

The development of parallel molecular dynamics algorithms has been only partially successful in addressing the need for simulations that extend over longer time periods. Molecular dynamics algorithms are relatively communication-bound and thus scale poorly once the number of atoms per processor drops below a certain point.[3,4] Applying parallel programming to these algorithms has been much more successful in increasing the size of the system that can be simulated and relatively unsuccessful in decreasing the amount of time required to simulate a system for a given time interval. To address these difficulties, theoreticians have developed free energy techniques and other sampling protocols that are designed to provide indirect estimates of the population of events that are rare on the time scale of the simulation.[5−9] Instead of trying to simulate rare events directly, a collection of shorter simulations is used to estimate the relative free energy of these rare events and therefore estimate their relative population.

If the individual simulations are small enough to fit on a single processor, then one method of speeding up these calculations using parallel architectures is to farm out each simulation to an individual processor. This is commonly referred to as an embarrassingly parallel algorithm. However, if the calculations are too large to fit on a single processor, then each simulation or task must be distributed over several processors, and the only way to run the calculation is to run

* Corresponding author e-mail: bruce.palmer@pnl.gov.

10.1021/ct600260u CCC: $37.00 © 2007 American Chemical Society
Published on Web 01/18/2007

many consecutive parallel tasks. This can be a major constraint if the individual tasks do not scale well above a certain number of processors. There is also no way to speed up the calculation if the number of processors available is more than the number of tasks, even if the individual tasks scale well beyond a single processor per task.

This paper will discuss the use of processor groups to achieve very high levels of scalability for algorithms that are based on multiple independent simulations. Processor groups are designed to allow the programmer to subdivide the domain of available processors (the "world group") into independent subdomains. Processor groups make it possible to run multiple concurrent independent calculations, each of which is running as a parallel, distributed task. This can be achieved with little additional effort beyond that required to write a conventional parallel code. These extensions have been incorporated into the Global Arrays Toolkit,[10] which has been used to obtain the results described below.

The remainder of this paper will give an overview of multiple task simulation strategies and then discuss the shared-memory programming model and the processor group extensions to it, the implementation of a spatial decomposition molecular dynamics algorithm using shared-memory programming, the development of a code that uses processor groups, and scaling results for the code.

Multiple Task Simulations

As discussed in the Introduction, multiple task simulations can be sped up in two ways using conventional parallel programming models. The first is to use an embarrassingly parallel approach to farm out tasks to individual processors; the second is to run multiple consecutive parallel tasks. For the embarrassingly parallel algorithm, the program starts with a list of independent tasks and assigns each processor a subset of tasks from the list. This is illustrated schematically in Figure 1a. A variant on this approach is the master−worker algorithm, where one processor (the master) assigns tasks to the remaining processors (the workers). As each worker processor finishes its task, it is assigned the next task on the list by the master. The embarrassingly parallel approach works very well if the individual tasks are relatively small calculations that can be run on single processors, but it breaks down for large simulations where each task must be distributed across more than one processor in order to be run at all. It also has the problem that there is no way to reduce the simulation time further once the number of processors exceeds the number of tasks.

Most general-purpose MD codes that perform free-energy calculations on parallel platforms use the approach of running consecutive parallel tasks. This is illustrated in Figure 1b. Each individual task is run in parallel, but only one task is run at a time. This approach works well if the individual tasks are too large to fit on a single processor but requires very good scaling of the parallel code to utilize large numbers of processors. This kind of scaling is difficult to achieve, however, and many molecular dynamics codes show very poor scaling once the number of atoms per processor drops to a few hundred.[3,4] Running consecutive parallel tasks is a natural extension of the way most parallel programs are written and can handle arbitrarily large systems, but it does not provide a good mechanism for utilizing large numbers of processors.

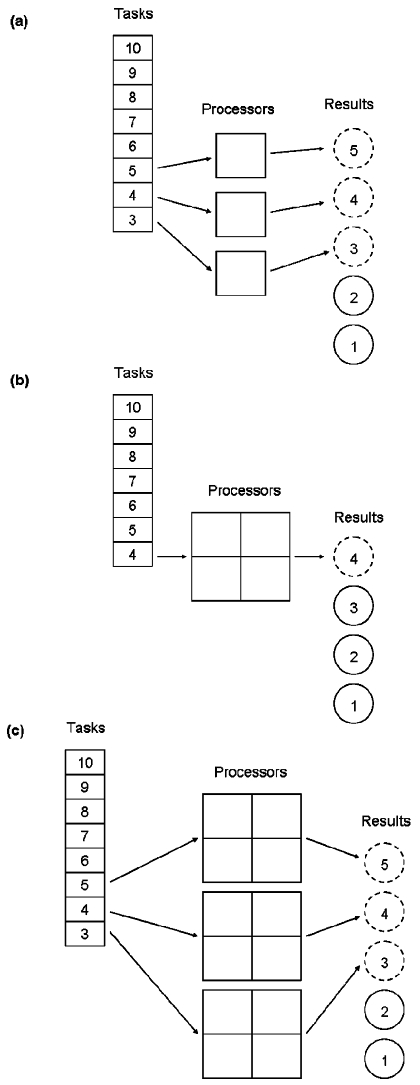

Figure 1. Schematic illustration of programming models discussed in the text: (a) embarrassingly parallel, (b) parallel sequential tasks, (c) concurrent parallel tasks using groups. Dotted circles indicate results that are currently being generated; solid circles represent completed tasks.

The approach discussed in this paper is to use processor groups to run multiple concurrent parallel tasks. This approach permits the use of very large numbers of processors to complete the calculation without taking the performance hit associated with poor scaling. The basic idea is to take the original collection of processors and subdivide them into smaller groups. Each group can then be used to run a parallel task that is independent of the other tasks. This is illustrated in Figure 1c. So far as the authors are aware, this approach has not been used in molecular dynamics, but it offers a way to greatly speed up calculations that consist of many independent tasks. The main disadvantage of this approach is that it is potentially much harder to program. However, by making use of the concept of a default processor group, discussed in more detail below, it turns out that a groups-based code is not significantly more complicated than a sequential parallel code.

Processor Groups and Shared Memory

The shared-memory programming model assumes memory can be divided into two categories, local and shared. Data structures that are created in local memory are visible only to the process that created them, while data structures in shared memory are visible to any process in the system. The message-passing programming model, on the other hand, assumes that all memory is local. Shared memory is a feature of some computer architectures in which multiple processors can directly access the same physical address space. Data objects created in this address space, such as arrays, can be written to and read from using a simple PUT/GET syntax that allows programmers to specify blocks of data in the shared array that need to be copied to or from local arrays. This style of programming tends to be much simpler than the distributed model used in message passing because shared memory preserves the global index space. Message passing, in contrast, requires the programmer to transform a data point from the global index space to a local index held on a particular processor. Communication in a shared programming model is accomplished by writing to and from arrays held in shared memory, while communication in a message-passing model is managed by sending data to a specific location on a specific processor. Although the shared-memory programming model originally reflected the physical layout of the memory on some parallel systems, it is also available on distributed memory platforms via the Global Arrays Toolkit.[10] This is a library of routines that allows programmers to create globally accessible data structures even if memory is completely distributed.

Most global operations in parallel programming are executed on the set of all available processors (the "world group"). These operations consist of synchronizations, global sums, and broadcasts. A synchronization forces all processes to wait until everyone has reached the same point in the execution before any process can proceed further, global sums add a value or values across all processors, and broadcasts send a value or values from one processor to all others. The global sum and broadcast operations implicitly synchronize the processors as well. Global operations are usually employed to guarantee data consistency across processors. Because they force processes to stop until everyone has reached the same point in the execution, global operations tend to degrade scalability in parallel codes, and their widespread use should be avoided. However, almost all parallel codes have at least a few global operations.

The presence of global operations is a major impediment to subdividing the world group into smaller groups that can run independent parallel tasks. Global operations in any task will force processes to wait until there is a corresponding global operation in other tasks. If the algorithm is such that other tasks will have different numbers or different sequences of global operations, the code will eventually hit a point where there is an unanswered global operation, and the program hangs. To avoid this, the concept of processor group has been introduced. Global operations executed on a processor group are restricted to only those processes in the group. It is also possible to restrict the scope of queries that return the number of processes and the local processor ID so that they act on subgroups instead of the world group.

For shared-memory programming models, processor groups also need to restrict the extent of shared arrays that are created in the context of a subgroup so that the address space represented by the shared array is accessible only from processes within the group. Implementing this functionality requires some modification of the memory allocation routines and additional layers of index translation, but this is transparent to the user. The main modification to the programming model is that programmers need to be able to specify that a given shared object is associated with a particular processor group, instead of an implicit assumption that it is being created on the world group. The Global Array Toolkit used in developing the cluster simulation application described here has been modified to include these capabilities.

Finally, an important concept associated with programming on processor groups is the default processor group. Support for this has been explicitly incorporated into the Global Arrays Toolkit, but the programming concept could also be used in codes based on message-passing interfaces (MPI), provided sufficient care is used. The default processor group is the group that all global operations and queries refer to unless another group is explicitly specified. For shared-memory programming, all shared data structures are created on the default group, again, unless explicitly specified otherwise. Conventional parallel programs have the world group as the default group. However, it is extremely useful to be able to specify some other group as the default. Parallel codes that have been written using the world group as the default can be repackaged as modules that run on a subgroup by wrapping them as a subroutine call. The calling program first creates the subgroup, sets the default group of all processes in the group to be the subgroup, and then calls the subroutine. The subroutine then runs exclusively on the subgroup. The advantage of this type of programming is that it allows modules to be developed as standard parallel codes before converting them to run on groups. This capability is explicitly supported in the Global Array Toolkit by subroutine calls that allow the programmer to set the default group, but the concept could also be executed using MPI by replacing explicit references to MPI_COMM_WORLD with a variable reference.

Parallel Molecular Dynamics Using Shared Memory-Style Programming

The parallel molecular dynamics code used in these studies is designed to simulate a Lennard-Jones fluid and fluid mixtures. There are no bonded interactions in this system, so there is no need for tables of bonded interactions and excluded atoms. The code is complicated by the need to incorporate a confining sphere that encloses one component of the Lennard-Jones mixture by forcing all atoms of the component to remain within a fixed distance of the center of mass of the cluster. This confining sphere uses a hard-sphere interaction to specularly reflect particles that cross the sphere boundary.

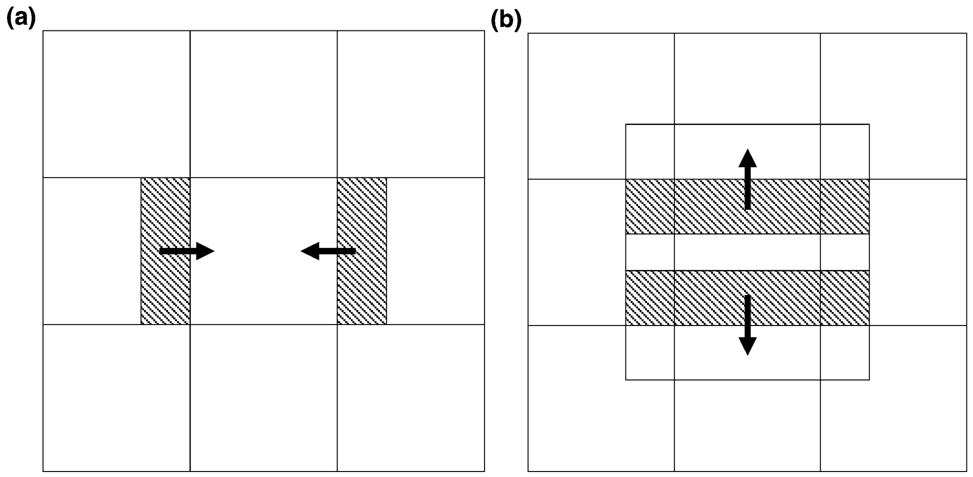

Figure 2. Schematic diagram of shift algorithm for a 2D system. (a) In the first stage, data that is within the boundary region of a neighboring east or west processor is sent to the neighboring processor. (b) In the second stage, data that is within the boundary region of a neighboring north or south processor is sent to the neighboring processor. This includes data that were transferred from east and west neighbors in the first stage.

The basic simulation kernel uses a spatial decomposition algorithm[11] to evaluate forces and distribute particle data. The original set of processors is decomposed into a three-dimensional block of processors (a set of eight processors would be divided into a cube of $2 \times 2 \times 2$ processors). The dimensions of the processor block grid are used to divide the simulation volume into a set of equal-sized cells. Particles in each cell of the simulation volume are then assigned to the corresponding processor. Forces in the Lennard-Jones fluid have a finite interaction length; that is, only particles that are within a finite cutoff distance R_c of each other have any interactions. To evaluate the forces at each time step, it is necessary to get the coordinates of all particles on surrounding processors that are within the cutoff distance. To reduce the number of data-transfer operations, it is usually desirable to pad the cutoff distance by an extra amount, δ. Thus, each processor must collect all particles that are within a distance $R_c + \delta$ of the spatial boundaries associated with the processor. This operation is accomplished using the shift algorithm.[11]

The shift operation consists of gathering particles that lie within a distance $R_c + \delta$ of the cell boundary along the x axis and sending their coordinates to the adjoining processors. This is illustrated in Figure 2a. The next step is to move all particles within a distance $R_c + \delta$ of the cell boundary along the y axis to adjoining processors. The particle locations from the adjoining cells along the x axis are already on the processor, so these can be sent along with particles located locally on the processor. This is illustrated in Figure 2b. After the transfer along the y axis is complete, the particles are transferred in a similar manner along the z axis. If all processors execute these transfers, then each processor will have a complete list of coordinates for all particles on neighboring processors within a distance of $R_c + \delta$ of the local cell boundaries. The fact that each of the updates is done sequentially along the axes reduces the total number of transfers to six, compared to the 26 transfers that would be required in three dimensions if data were collected from each of the neighboring processors independently. The shift algorithm generally needs to be performed twice during each time step. The first shift is before the force calculation when all particle coordinates must be exchanged. The second shift occurs after the force calculation, when all of the partial forces must be sent back to their home processors. This second shift can be eliminated by dropping the use of Newton's second law on all interactions between a locally held particle and a remotely held particle, but this can significantly increase the time spent in the force calculation.

The data transfer must be repeated at every time step. At periodic intervals, it is also necessary to redistribute the particles so that particles that have drifted outside the box associated with a given processor are assigned to new processors. Because the cutoff distance is padded by the amount δ, particles that have drifted a short distance outside the simulation cell do not cause any problems with the force calculation. Eventually, however, particles will move far enough that interactions will be missed if the particles are not reassigned to their appropriate processors. This re-assignment is performed after a fixed number of steps. To summarize, the major communication steps in this algorithm are (1) the distribution of particles to processors on the basis of current coordinates, (2) the gathering of particle coordinates before evaluating forces, and (3) the scattering of partial forces back to particles after the force calculation. The bulk of the communication is from steps 2 and 3, which must be performed at every time step; the particle redistribution step occurs less frequently. Additional communication steps are associated with initialization and closeout of the calculation, but these are typically an insignificant fraction of the total execution time. There are also synchronizations and global summations of small quantities of data.

Individual shifts can be implemented using either message passing or using a shared-memory programming model. The message-passing model assumes point-to-point communication in which data are sent from a local array on one processor to a local array on another processor. Typically, this requires that both the sending and receiving processes post signals that they intend to send or receive a message, and they must also specify the message length. Since the length is generally not known beforehand, this requires the exchange of an extra integer before the actual list of coordinates and so forth can be sent. Instead of exchanging individual messages, the shared-memory model creates a set of shared memory or otherwise globally accessible arrays that are used for communication. The shared memory arrays are either one-dimensional or quasi-one-dimensional (the second dimension is typically small). To communicate, two arrays are needed. The first array, referred to as the size array, is one-dimensional and only contains a number of elements equal to the number of processors. Each processor owns one data element that is used to store the number of particles that need to be moved. The second array, referred to as the communication array, is much larger. For concreteness, assume that the array will be used for moving coordinates around. A parameter is chosen that represents the maximum number of atoms that can be expected to reside in the communication array for any one processor at any point in the communication cycle. Call this parameter MAXAT. The size of the communication array for moving around a vector quantity such as the coordinates is then NPROCS $\times$ MAXAT $\times$ 3, where NPROCS is the total number of processors. Each processor then owns a MAXAT $\times$ 3 sized block of the total communication buffer.

A communication step takes place in two stages. In the first stage, all processors determine which particles need to have their coordinates sent to a neighboring processor. These coordinates are all gathered into a local buffer and then copied into the portion of the communication array owned by that processor, using a PUT operation. The number of particles that each processor puts in the communication array is also placed in the size array. After these operations are complete, a synchronization call is made to guarantee that all data is finished being copied to the communication array and the size array. The second stage begins by having each processor get the number of elements in the size array from the appropriate adjoining processor using a GET operation. Once this number has been obtained, the processor can get the block of data corresponding to the adjoining processor from the communication array. The entire sequence of operations is illustrated schematically in Figure 3 for a 2 $\times$ 2 array of processors. Each processor is receiving data from the processor to its right. Periodic boundary conditions are assumed, so each processor at the end of a row wraps around and gets data from the first processor in the row.

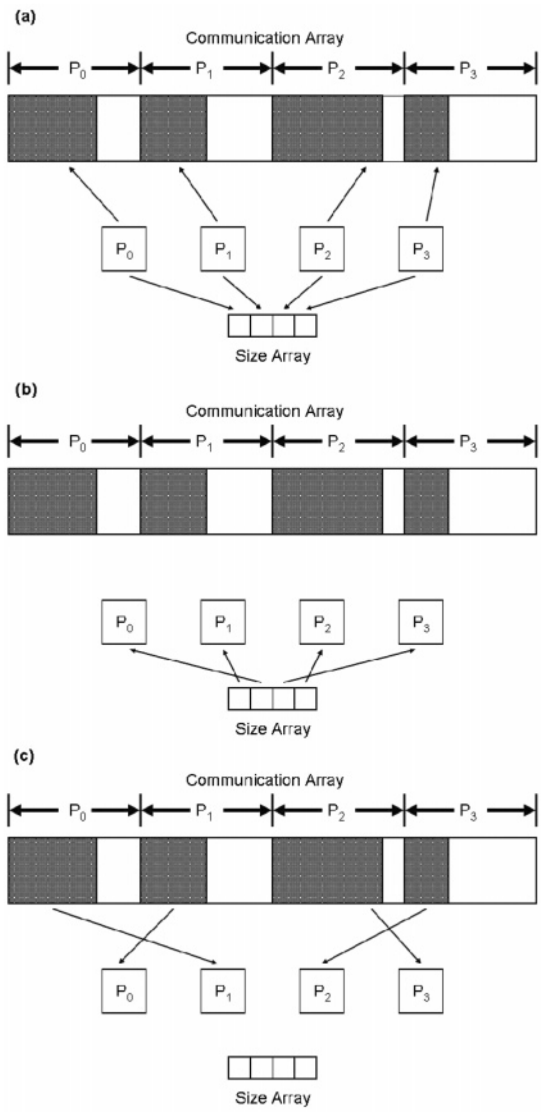

Figure 3. Schematic illustration of particle exchange for a four processor calculation which is decomposed into a 2 $\times$ 2 grid. (a) Each processor copies particle coordinates and so forth that need to be transferred to another processor to its locally held portion of a communication array. It also copies the number of particles that are in the communication array to a size array. (b) After a synchronization operation, each processor gets the amount of data from neighboring arrays from the size array. (c) Each processor gets the data for a neighboring process from the communication array.

Statistical Mechanics of Cluster Nucleation

The multiple task simulation targeted in this paper is the evaluation of atomic cluster size distribution functions for clusters of an immiscible fluid in a solvent. These calculations are not traditional free-energy calculations but share the feature of consisting of multiple independent simulations. The distributions that result from the individual simulations are converted into free-energy profiles for further analysis. The profiles lead directly to cluster dissolution rates. When combined with additional simulations of the relative free energies of clusters with different numbers of particles, they can also give the cluster nucleation rates. When such information is obtained for all cluster sizes up to the critical cluster (i.e., the cluster that defines the peak in the work of formation), a complete picture of the nucleation of phase separation can be developed.

A parallel single-task MD code was written that is designed to evaluate the cluster size probability distribution function

$$P(V) \propto \exp(-\beta p_{\mathrm{ext}} V) \int \mathrm{d}\mathbf{r}^N \exp[-\beta U(\mathbf{r}^N)] \prod_{i=1}^{M} \theta(r_i - r_{\mathrm{cnf}})$$

where $\mathbf{r}^N$ is a collective coordinate representing the location of N particles, U is the potential energy of those particles, β

$= 1/k_BT$ is the inverse temperature, $V = 4\pi r_{cnf}{}^3/3$ is the volume of the confining sphere, r_{cnf} is the corresponding radius of the confining sphere, r_i is the distance of a particle to the cluster center of mass, $\theta(x)$ is the Heaviside step function, and the product runs over a subset containing M particles. The external pressure p_{ext} is an arbitrary applied pressure that guarantees that the probability distribution decays to zero at large volumes. This distribution can be directly related to the evaporation (or dissolution) rate of a cluster of size M.[12, 13] For evaporation, the system is chosen so that $M = N$, but for dissolution, the system would be configured so that $M < N$. Once evaluated, $P(V)$ can be converted to a free energy, $A(r_{cnf})$, using the expression[12]

$$A(r_{cut}) = -k_BT\ln[P(V)] - p_{ext}V = -k_BT\ln[P(r_{cut})/(4\pi r_{cnf}{}^2)] - p_{ext}V$$

The second expression is frequently more useful since the Monte Carlo sampling actually generates the distribution $P(r_{cnf}) = 4\pi r_{cnf}{}^2P(V)$. Note that the explicit dependence of $A(r_{cnf})$ on p_{ext} drops out in this expression. The monomer dissolution rate of the cluster is directly proportional to the derivative of A with respect to r_{cnf}.

The problem of evaluating the distribution $P(V)$ can be reduced to simulating the behavior of a collection of particles inside a confining sphere of radius r_{cnf}. For simplicity, only the case in which all particles are contained in the confining sphere is discussed; the extension to the case where a subset of particles is restricted to the confining sphere is straightforward. The distribution $P(V)$ has been simulated previously using Monte Carlo techniques.[12,13] In this paper, the use of molecular dynamics simulations to evaluate $P(V)$ is explored. The confining sphere can be considered as an infinitely hard potential whose center corresponds to the center of mass of the confined particles. Whenever a particle hits the confining sphere, it is reflected specularly away from the sphere surface. The velocity of the remaining particles is also altered slightly because the momentum of the center of mass of the remaining particles is specularly reflected by the collision as well. A more detailed description of how these collisions are implemented will be provided elsewhere; for the purposes of this paper, it is important to note the following points:

(1) The presence of “hard-sphere”-type collisions means that time steps during which a collision occurs must be broken up into smaller intervals. Before and after the collision, the trajectories are smooth and can be integrated using a conventional second-order algorithm. The velocity−Verlet algorithm, recast as a second-order Gear predictor−corrector,[14] is especially convenient because it only needs the current coordinates and velocities to execute the next step. After determining that a collision with the confining sphere occurred during a time step, the time at which the collision occurred is calculated and the original step is broken up into pre- and postcollision intervals. The system is updated using a time step equal to the precollision interval, and then the velocities are modified using the rules for hard-sphere dynamics. Once the postcollision velocities are calculated, a second step is taken using a time step equal to the postcollision interval. If another collision is detected within the second interval, then the process is repeated until the original time step has been completed.

(2) For parallel simulations, hard-sphere dynamics associated with the confining sphere add several additional global operations. These are needed because each processor independently determines whether one of its particles has hit the confining sphere and, if so, when. This information must then be shared with all other processors. Additionally, processors also check for special cases, such as multiple collisions within a single time step and trajectories that loop into and out of the confining sphere within a time step, and communicate the results of these checks to other processors. Although these special cases occur infrequently, they cause significant problems if not handled correctly.

(3) The confining sphere radius is chosen using a conventional Monte Carlo procedure.[12] After a fixed number of steps, a new confining sphere volume is generated. If the new volume is smaller than the old volume, then a check is performed to see if any of the particles in the cluster have moved outside the confining sphere. If they have, the volume is rejected. If no particles are outside the sphere, then the sphere is accepted or rejected using a standard Monte Carlo acceptance test based on the change $p_{ext}\Delta V$. This test also adds an extra layer of synchronization.

Overall, the simulations proceed much like conventional molecular dynamics simulations. The main differences are that the coordinate update steps are more complicated and involve much higher levels of synchronization.

The system investigated in this study consists of a two-component Lennard-Jones fluid characterized by the potential function

$$U(\mathbf{r}^N) = \sum_{i<j}\varphi_{ij}(r_{ij})$$

where the Lennard-Jones interaction φ_{ij} has the form

$$\varphi_{ij}(r_{ij}) = \begin{cases} 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] - \varphi^0_{ij} & r_{ij} < R^c_{ij} \\ 0 & r_{ij} \geq R^c_{ij} \end{cases}$$

The constant $\varphi_{ij}{}^0$, representing the cutoff interaction energy, is

$$\varphi^0_{ij} = 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{R^c_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{R^c_{ij}}\right)^{6}\right]$$

This choice guarantees that $\varphi_{ij}(r)$ goes continuously to zero as r approaches R^c_{ij}. For these simulations, the fluid contains two components denoted by A and B. For all interactions, σ_{ij} and ϵ_{ij} are set equal to 1. The particle masses are also set to 1. If both i and j belong to the same component, then R^c_{ij} is set equal to 2.5. If i and j belong to different components, then R^c_{ij} is set equal to $2^{1/6}$. These choices guarantee that interactions between particles in the same component are attractive at longer distances while interactions between particles in different components are purely repulsive. The pair interactions are illustrated in Figure 4. For these simulations, the component forming the cluster is B (referred to hereafter as the solute) while A is the majority component

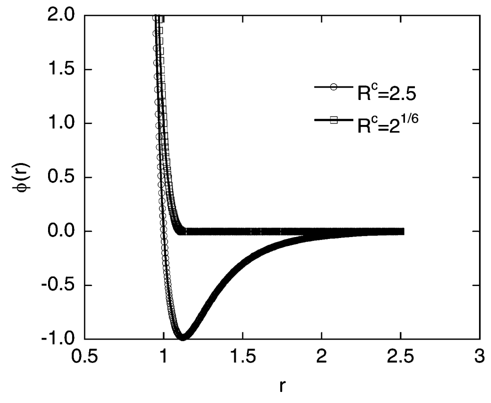

Figure 4. Plot of Lennard-Jones interaction potential using both long- and short-range cutoffs.

and acts as the solvent. The repulsive cross interaction means that B will tend to form clusters within A.

The output of the simulation is the distribution $P(r_{\mathrm{cnf}})$ [which is readily converted to $P(V)$]. This is obtained by binning the corresponding values of the radius of the confining sphere that are selected at each Monte Carlo step. The program has also been modified to incorporate an equilibration protocol into each trajectory along with an additional setup routine that creates an initial condition by specifying the number of Lennard-Jones particles of each type that are used in the simulation. The initial conditions are chosen so that the minority B component is always located in a roughly spherical blob at the center of the simulation cell. To improve convergence, the range over which the confining sphere radius is sampled is restricted. The simulation is initially run for a short period during which the confining sphere radius is brought within the sampling range, it is then equilibrated further. After equilibration, data is collected and the distribution $P(r_{\mathrm{cnf}})$ is evaluated. The data collection stage is much longer than the equilibration stage, primarily because the trajectory has to be run for a very long time to get good statistics for $P(r_{\mathrm{cnf}})$.

Results

The scaling properties of the groups-based simulation code were investigated by first determining the scaling behavior of the isolated molecular dynamics module running as a standard parallel application. After determining the single-task scaling behavior, the scaling properties of multiple tasks running on groups were evaluated. The single-task scaling behavior was determined for a system consisting of 4000 solvent particles and two solute particles. The simulations were run for 50 000 steps using a time step of $\Delta\tau = 0.01$. These simulations are too short to provide valid scientific results, but they are long enough to characterize the scaling behavior of the code. Longer simulations that produced useful data are discussed below. The simulations were run using the isothermal/isobaric method of Nosé and Anderson[15,16] with a reduced temperature of 0.8 and a reduced pressure of 0.05. This locates the simulations in the liquid part of the Lennard-Jones phase diagram for the pure Lennard-Jones fluid.[17] The single-task scaling behavior of the molecular dynamics module on an HP dual Itanium2 1.5 GHz cluster with a Quadrics switch using the Elan communication

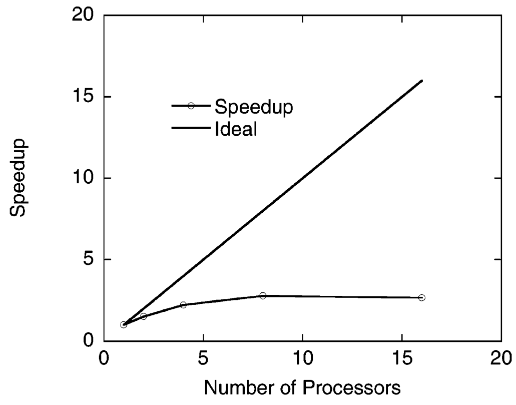

Figure 5. Single task scaling behavior of molecular dynamics simulation.

libraries is shown in Figure 5. It can be seen that the scaling behavior of this calculation is fairly poor and drops off rapidly as the number of processors increases. The speedup from one to two processors is reasonable but starts falling off significantly in going from two to four processors. Beyond that, the speedup is relatively flat, and at 16 processors, the speedup has actually started to decrease. The poor scaling for this calculation is primarily due to the relatively small problem size and very poor load balancing associated with the initial condition. Simulations run for longer periods of time have better load balancing as the density equilibrates and density fluctuations present at the beginning disappear.

To perform multiple, concurrent parallel tasks, the molecular dynamics code was modified to divide the available processors in the world group into subgroups. The code then executes the tasks on the groups. When a task is completed on one group, the group gets the next task from a global counter. The global counter is implemented using the READ-INCREMENT functionality available in the Global Arrays Toolkit. A shared array, consisting of a single integer is created at the start of the calculation and initialized to zero. The READ-INCREMENT function reads the current value of an array variable and increments it by 1. As each processor group completes a task, the processor corresponding to process 0 in the group performs a read-increment on the array variable acting as the global counter and translates the current value of the variable into a set of task parameters.

Simulations of M tasks were run with each task representing a system of 4000 solvent particles and between two and $M + 1$ solute particles. For the timing studies described below, the number of tasks was set to 512. Most of the remaining parameters are the same as for the calculations described in the single-task scaling study, except that the number of time steps was reduced to 10 000 in order to reduce the overall simulation time. Because the tasks with larger numbers of solute particles were expected to take slightly longer, the tasks were executed in reverse order to improve load balancing. These simulations were run on between eight and 1024 processors on the HP cluster described above. On the basis of the single-task simulation results, the available processors were divided into groups of two processors. The results are shown in Figure 6.

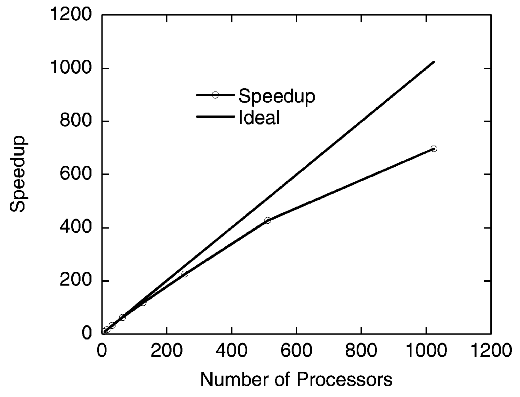

Figure 6. Speedup curve for 512 tasks as a function of the number of processors. The available processors were divided into groups containing two processors each.

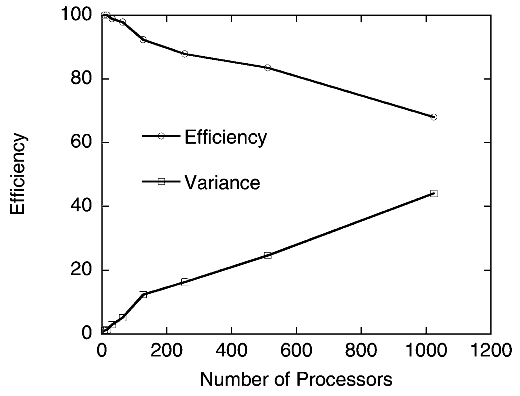

Figure 7. Parallel efficiency and maximum variance in execution times on each of the groups as a function of the number of processors for 512 tasks.

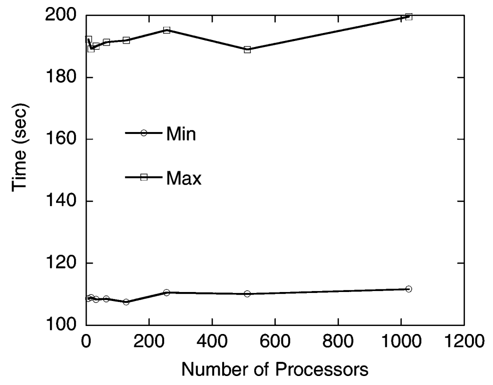

Figure 8. Plot of minimum and maximum task execution times for 512 tasks.

The results show good speedup all the way up to 1024 processors. The parallel efficiency, which is the actual speedup divided by the theoretical maximum speedup, is shown in Figure 7. The efficiency remains above 80% all the way up to 512 processors and is close to 70% even for 1024 processors. The main reason for the drop in efficiency appears to be the granularity of the execution times of the individual tasks. As the number of tasks per group drops, the variation in the amount of time required for each group to finish its tasks increases.

The variation in individual task execution times can be seen in Figure 8, where the minimum and maximum execution times for the individual tasks are plotted as a function of the number of processors. If the individual tasks are truly running independently, then each task should take about the same amount of time regardless of the total number of processors. This further implies that the minimum and maximum execution times for the collection of tasks should also be independent of the total number of processors, and this is seen in Figure 8, although there may be a small increase in execution times at very large processor counts. The figure also shows that there is a substantial variation in execution times. The maximum execution time is over 60% higher than the minimum execution time. This variation is much larger than the variation in the total number of particles in the different tasks (approximately 13%) and is due to the relatively short execution times and differences in short-time behavior caused by differences in the initial configuration.

The variance in execution time on individual groups was also calculated and is plotted along with the parallel efficiency in Figure 7. If T_i represents the total time spent on group i executing its tasks and T_{min} and T_{max} are the minimum and maximum values of T_i across the different groups, then the variance is $100(T_{\mathrm{max}} - T_{\mathrm{min}})/T_{\mathrm{max}}$. Note that T_{max} is almost identical to the overall execution time for the entire calculation. The variance is inversely correlated with parallel efficiency; as the variance increases, parallel efficiency decreases. This supports the conclusion that the main source of the decrease in parallel efficiency is the granularity of the individual tasks. This problem would probably decrease for longer simulations where the variation in initial conditions is not as important. For longer simulations, the execution times should track more closely with the total number of particles, and the execution times would be much closer together.

The short calculations described above were good only for evaluating scalability of the code. To produce useful scientific data, much longer calculations are needed. To demonstrate that such calculations are possible, a set of simulations consisting of 10^8 steps were run on systems containing two to five solute particles. In order to run simulations containing such a large number of steps in a reasonable amount of time, the number of solvent particles was reduced to 400. For the small solute clusters investigated in this study, 400 solvent particles should be sufficient. Simulations in this regime indicate that the correlation length of the radial distribution function for the pure Lennard-Jones fluid is on the order of $3-4\sigma$, while the typical dimension of the simulation cell is a little over 8σ. The first 2 000 000 steps were used to move the confining sphere to within the maximum value of 4.0σ and to equilibrate the system. The remainder of the simulation was used to collect data. The simulations were run on eight processors divided into four groups of two processors each. Some shorter runs on trajectories containing 1 000 000 steps indicate that the speedup in going from one to two processors is about 1.6. The total execution time for all four calculations was about 52 h, so the use of groups amounted to a substantial decrease in the overall execution time. There is no additional benefit to using groups containing more than two processors for a system this small since the speedup actually starts to decrease

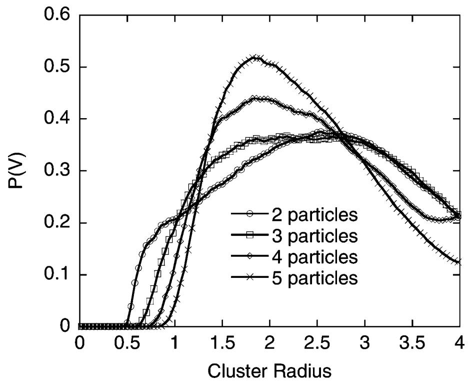

Figure 9. Confining sphere size distributions for simulations containing two to five solute particles and 400 solvent particles.

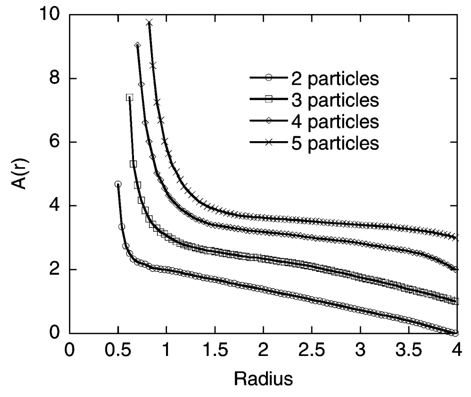

Figure 10. Cluster free energies as a function of size for simulations containing two to five solute particles and 400 solvent particles.

at four processors. For these longer simulations, the load imbalance issues are much smaller. The total variation in execution times for the different tasks is about 1.6%.

The probability distributions for the confining sphere radius are shown in Figure 9. A few things are worth noting. First, the leading edge of the distributions moves to larger radius values as the number of solute particles increases. This is expected because more particles imply a larger cluster. Second, even with 10^8 steps, the distributions still appear to contain noticeable amounts of noise. This is a common problem with simulations containing many degrees of freedom where the focus is to determine the behavior of a single coordinate. It is difficult to obtain accurate averages since each point in the distribution must be sampled many times but at the same time a substantial amount of work is required for each system update.

The actual quantity needed for evaluating dissolution rates is the free-energy curve $A(r_{\mathrm{cnf}})$, which can be derived from the distributions $P(V)$. The curves for $A(r_{\mathrm{cnf}})$ are shown in Figure 10. Because the $A(r_{\mathrm{cnf}})$'s are proportional to the logarithm of the distribution functions, much of the noise in the $P(V)$ curves is suppressed. It is possible that, after applying some smoothing algorithms to these curves, reasonable values of both the derivative of $A(r_{\mathrm{cnf}})$ and the location of the minimum value of the derivative could be obtained and used to evaluate the dissolution rate. A more extensive analysis of these results and results for larger clusters will be presented elsewhere.

As a final comment on the performance of multitask codes using groups, if only four processors had been available, the most efficient way to perform the simulations would be to run on groups containing only a single processor. This would be equivalent to the embarrassingly parallel algorithm. The reason for using only one processor per group is that efficiency is highest for single-processor calculations, and since the number of tasks exceeds the number of processors, the shortest time to solution will be obtained by running each task at the highest possible efficiency. This is generally true for any calculation with a limited number of processors. The shortest time to solution is obtained by running individual tasks on the smallest number of processors on which the calculation will fit. It is only when a large number of processors are available so that *all* tasks can be run concurrently that there is a time benefit to increasing the number of processors in the group.

Conclusions

The results presented here demonstrate that processor groups can be used to substantially reduce the amount of time required to perform multiple independent molecular dynamics simulations. The scenario of many independent calculations is often encountered in evaluations of free energies, which are an important class of calculations in biochemical simulations and other evaluations of chemical stability and reactivity. The results of these scaling studies indicate that processor groups can be used quite successfully to run multiple concurrent instances of a parallel calculation. The results also illustrate the utility of the default processor group concept, which allows developers to easily reconfigure parallel applications to run on groups with only minor adjustments to the code.

For free-energy calculations, processor groups can be used to either run multiple concurrent simulations when each individual simulation is too large to fit on a single processor or can be used to speed up simulations in which the number of available processors exceeds the number of individual simulations. Without processor groups, the only available alternatives are to run individual simulations on single processors following an embarrassingly parallel model or to submit multiple independent calculations to the computer. The first option can only be used if the individual simulations fit on a single processor, and the second option, besides being inconvenient, breaks down completely if more complicated simulation protocols are used in which initial results are used to bias the sampling and improve statistics.

Acknowledgment. This work was supported by Pacific Northwest National Laboratory (PNNL) and the U.S. Department of Energy's Office of Science through the William R. Wiley Environmental Molecular Sciences Laboratory (EMSL) and the Division of Chemical Sciences and was performed in part using the Molecular Science Computing Facility (MSCF) in EMSL. The MSCF and EMSL are funded by the Office of Biological and Environmental Research in the

U.S. Department of Energy. Pacific Northwest National Laboratory is operated by Battelle Memorial Institute for the U.S. Department of Energy under Contract No. DE-AC06-76RLO 1830.

References

(1) Wang, W.; Donini, O.; Reyes, C. M.; Kollman, P. A. Biomolecular Simulations: Recent Developments in Force Fields, Simulations of Enzymatic Catalysis, Protein−Ligand, Protein−Protein, and Protein−Nucleic Acid Noncovalent Interactions. *Ann. Rev. Biophys. Biomol. Struct.* **2001**, *30*, 211.

(2) Garrett, B. C.; Schenter, G. K.; Morohita, A. Molecular Simulations of the Transport of Molecules across the Liquid/Vapor Interface of Water. *Chem. Rev.* **2006**, *106*, 1355.

(3) Phillips, J. C.; Braun, R.; Wang, W.; Gumbart, J.; Tajkhorshid, E.; Villa, E.; Chipot, C.; Skeel, R. D.; Kalé, L.; Schulten, K. Scalable Molecular Dynamics with NAMD. *J. Comput. Chem.* **2005**, *26*, 1781.

(4) Germain, R. S.; Zhestkov, Y.; Eleftheriou, M.; Rayshubskiy, A.; Suits, F.; Ward, T. J. C.; Fitch, B. G. Early Performance Data on the Blue Matter Molecular Simulation Framework. *IBM J. Res. Dev.* **2005**, *49*, 447.

(5) Straatsma, T. P.; McCammon, J. A. Multiconfiguration Thermodynamic Integration. *J. Chem. Phys.* **1991**, *95*, 1175.

(6) Tobias, D. J.; Brooks, C. L;, Fleischman, S. H. Conformational Flexibility in Free Energy Simulations. *Chem. Phys. Lett.* **1989**, *156*, 256.

(7) Tobias, D. J.; Brooks, C. L. Calculation of Free Energy Surfaces using the Methods of Thermodynamic Perturbation Theory. *Chem. Phys. Lett.* **1987**, *142*, 472.

(8) Widom, B. Some Topics in the Theory of Fluids. *J. Chem. Phys.* **1963**, *39*, 2808.

(9) Straatsma, T. P. Free Energy by Molecular Simulation. *Rev. Comput. Chem.* **1996**, *9*, 81.

(10) Nieplocha, J.; Palmer, B. J.; Tipparaju, V.; Krishnan, M.; Trease, H. E.; Apra, E. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *Int. J. High Perf. Comput. Appl.* **2006**, *20*, 203.

(11) Plimpton, S. Fast Parallel Algorithms for Short-Range Molecular-Dynamics. *J. Comput. Phys.* **1995**, *117*, 1.

(12) Kathmann, S. M.; Schenter, G. K.; Garrett, B. C. Dynamical Nucleation Theory: Calculation of Condensation Rate Constants for Small Water Clusters. *J. Chem. Phys.* **1999**, *111*, 4688.

(13) Schenter, G. K.; Kathmann, S. M.; Garrett, B. C. Variational Transition State Theory of Vapor Phase Nucleation. *J. Chem. Phys.* **1999**, *110*, 7951.

(14) Allen, M. P.; Tildesley, D. J. *Computer Simulation of Liquids*; Oxford University Press: New York, 1987.

(15) Andersen, H. C. Molecular Dynamics Simulations at Constant Pressure and/or Temperature. *J. Chem. Phys.* **1980**, *72*, 2384.

(16) Nosé, S. A. Unified Formulation of the Constant Temperature Molecular Dynamics Methods. *J. Chem. Phys.* **1984**, *81*, 511.

(17) Hansen, J.-P.; Verlet, L. Phase Transitions of the Lennard-Jones System. *Phys. Rev.* **1969**, *184*, 151.

CT600260U