Adaptive Load Balancing for Long-Range MD Simulations in A Distributed Environment

Sumanth J.V, David R. Swanson, Hong Jiang
Department of Computer Science and Engineering
University of Nebraska-Lincoln, U.S.A
{sumanth, dswanson, jiang}@cse.unl.edu

Abstract

Molecular Dynamics, a computationally intensive application is used by researchers in various fields. The inherent parallelism [13] in the computations involved with this application can be exploited in parallel and distributed environments. However, in distributed environments such as the Grid [6], the available resources, namely the network and computational power, are continually changing with respect to every available node. To optimally utilize these dynamic resources, a scheduler should be able to continually adapt to the changes and suitably vary the load scheduled to every available node. We propose one such scheduling algorithm. The proposed scheduling algorithm builds and continually updates a model of the distributed system, which it then uses to make decisions about how to optimally redistribute the load in the system at every time step of the MD simulation. The scheduling algorithm can additionally handle dynamic changes in the number of nodes available for computation at runtime. We then demonstrate the efficiency of our scheduling algorithm when applied to MD simulations in a distributed environment.

1 Introduction

Molecular Dynamics (MD) is a powerful technique used to obtain static or dynamic properties of liquids and solids. It can more formally be defined as a computer simulation technique where the time evolution of a set of interacting atoms is followed by integrating their equations of motion [2]. There are numerous applications for MD simulations in diverse fields of science and technology such as chemistry, astronomy, biophysics, solid-state physics, material science and fluid dynamics to mention a few.

MD simulations are not very memory intensive. Their space complexity grows linearly with the number of atoms being simulated. However, their time complexity grows quadratically with the number of atoms being simulated. Being a very computationally intensive application [15], various solutions to improve execution times have been investigated. The most common methods for improving performance are parallelization[5] and using custom hardware [16].

With the advent of technologies such as the Grid [6], a very large number of heterogeneous nodes are becoming available for researchers to run large jobs on. To be able to utilize such resources efficiently, an adaptive load balancing mechanism is essential. The reason for requiring the load balancing to be adaptive is that the available computational power on each of the available nodes is not constant. Further, even the available network bandwidth and memory latencies can vary dramatically over time.

In this paper, we design and implement an adaptive load balancing scheduler for long range molecular dynamics. Our scheduler determines the number of atoms to assign to each client node based on its performance during the previous time step. Our solution is capable of taking into consideration the dynamic variations in the available computational power available and network bandwidth and latency with respect to every node taking part in the computation. It can also handle client nodes becoming available at arbitrary times during the MD simulation. Our current implementation does not handle node failures yet, however, once an efficient way to detect node failure is determined, our scheduling algorithm will be capable of handling it efficiently.

Despite using our algorithm to schedule MD in a distributed environment in this paper, the proposed scheduler can also be used for a wide range of computational problems such as numerous matrix operations, image processing algorithms, partial differential equation solvers etc.

Some of the techniques previously used to load balance MD simulations include the orthogonal recursive bisection [3] [7], hashed oct-tree [18], costzones [9], cell redistribution [1] and morton-ordering [11]. However, all of these techniques take into consideration the spatial distribution of the atoms in the system since they they assume that the spatial decomposition method is used for parallelization. In this paper, we focus on load balancing the atom-decomposition method. The performance of this method is independent of the spatial distribution of the atoms in the system. This method is commonly used for long-range MD and MD simulations of very dense systems.

The rest of this paper is organized as follows: In section 2, we describe the basic aspects of an MD simulation, then in section 3, we detail our framework for deploying MD in a distributed environment, in section 4, we describe the model that we use to predict the performance of the distributed system, in section 5 we formulate how the system model can be used to determine an optimal schedule, in section 6, we describe our scheduling algorithm, in 7 we determine a few enhancements to the proposed scheduler that allows it to improve the utilization of available resources, in section 8, we evaluate the performance of our scheduling algorithm, in section 9 we list possible directions for future work and we conclude the paper in section 10.

2 Computational Aspects of MD Simulations

2.1 Basic Equations

The computational task of an MD simulation [2] is to perform the time integration of the differential equation (1), (2) with given initial atom positions and velocities i.e. $\{\overrightarrow{r}_i(0), \overrightarrow{v}_i(0) | i = 1, 2, \ldots, N\}$ and obtain the positions and velocities at a later time i.e. $\{\overrightarrow{r}_i(t), \overrightarrow{v}_i(t) | i = 1, 2, \ldots, N\}$.

$$\frac{\partial^2 \overrightarrow{r}_k(t)}{\partial t^2} = \overrightarrow{a}_k(t) \tag{1}$$

$$= \sum_{i<j} \overrightarrow{r}_{ij}(t) \left(-\frac{1}{r} \frac{\partial u(r)}{\partial r} \right) \Bigg|_{r=r_{ij}(t)} \cdot (\delta_{ik} - \delta_{jk}) \tag{2}$$

where,

$$\delta_{ik} = \begin{cases} 1, i = k \\ 0, i \neq k \end{cases} \tag{3}$$

is the kronecker delta function and $u(r)$ is the potential function.

Forces are computed as the negative gradient of the potential as

$$F_k = -\frac{\partial V(\overrightarrow{r}^N)}{\partial \overrightarrow{r}_k} = -\left(\frac{\partial V}{\partial x_k}, \frac{\partial V}{\partial y_k}, \frac{\partial V}{\partial z_k} \right) \tag{4}$$

where $V(\overrightarrow{r}_k) = \sum_{i<j} u(\overrightarrow{r}_{ij})$ and $\overrightarrow{r}_{ij} = \overrightarrow{r}_i - \overrightarrow{r}_j$.

2.2 Potential Function

The long range potential function that we use in our simulation is the Stockmayer potential [4]. The Stockmayer potential is similar to the Lennard-Jones potential [8] except that it adds an inverse third power term to model strong electrostatic contributions in polar molecules like ammonia and water vapor. The potential function is described by equation 5

$$u(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} - \delta \left(\frac{\sigma}{r}\right)^{3} \right] \tag{5}$$

where, δ is a dimensionless constant that measures the polarization of a substance, ϵ and σ (which have dimensions of energy and length, respectively) are constants characteristic of the chemical species of the colliding atoms, and r is the inter-atomic separation.

2.3 Time Integration

We use the velocity-verlet [17] algorithm to perform the time integration of the forces. It has moderate short term energy conservation. But there is very little long term energy drift. The algorithm is based on equations 6 and 7.

$$r(t + \delta t) = r(t) + v(t) \cdot \delta t + \frac{1}{2} \cdot a(t) \cdot \delta t^2 \tag{6}$$

$$v(t + \delta t) = v(t) + \frac{1}{2} \cdot [a(t) + a(t + \delta t)]\, \delta t \tag{7}$$

2.4 Implementation

The general algorithm for a MD simulation is illustrated in (Fig. 1). Since we are using a pair-potential, computation of the forces requires the calculation of N^2 interactions inside a double nested loop.

```
1: Read initial positions and velocities of all the atoms.
2: Use a specified potential interaction $V$, a function of
   atom positions.
3: for every time-step do
4:     Compute the non-bonded forces on every atom $i$,
       $F_i = -\frac{\partial V}{\partial r_i} = \sum_j F_{ij}$.
5:     If required, compute forces due to bonded interactions, restraining forces and external forces.
6:     Compute the kinetic and potential energies.
7:     Update the configuration of the system by numerically integrating Newton's equations of motion.
8:     If required, output positions, velocities, accelerations, energies, temperature, pressure, etc.
9: end for
```

Figure 1. General MD algorithm

We use the atom-decomposition algorithm [12] for parallelization since the faster spatial-decomposition [13] techniques work well only with short-range potentials. The atom decomposition method assigns each node a subset of the atoms. The assignment of atoms to the nodes remains the same irrespective of the atoms physical location at any time step in the simulation. Each node computes the forces and updates the positions and velocities of its assigned atoms and hence the name atom-decomposition. At the start of every time step, all the atomic positions have to be distributed among all the nodes.

One can imagine an $N \times N$ force matrix where the $(i, j)^{th}$ entry represents the force exerted by atom j on atom i. This matrix is dense if a long range potential is used. It is also skew symmetric due to Newton's third law i.e. $F_{ij} = -F_{ji}$ for a pair-potential. In an actual implementation, this is just stored as a 1-dimensional array whose elements are the sum of the forces on each atom.

3 Distributed framework for MD

We assume a very heterogeneous distributed environment such as a condor pool [10] for our MD simulation. The available computational power even on a single node in such an environment can vary significantly over time. It is the responsibility of our proposed scheduler to track such changes and optimize the number of atoms assigned to the client nodes for force computation. Further, in such an environment, the number of clients available to take part in the computation also varies over time.

We use a client-server model for deploying MD in a Grid. The server is comprised of a control thread for controlling the entire simulation as shown in Fig. 3, a server thread which spawns a new client handler for every client that connects to the server and client handler threads as shown in Fig. 4 that handle communication with each client node. The primary role of the server is distributing the atoms to all the clients and load balancing the work assigned to each client. A client waits for atom positions from the server and computes the forces on the assigned atoms and returns these forces to the server.

```
1: Establish a connection with the server.
2: **while** true **do**
3:     Wait for positions of all $N$ atoms from the server.
4:     Compute forces on the assigned $n_i$ atoms.
5:     Return the forces computed and the time taken to do
       so to the server.
6: **end while**
```

Figure 2. Client Algorithm

The server's control thread is responsible for reading the initial positions and velocities of the atoms from a file and distributing all the initial positions to all the clients. It is also responsible for determining how many atoms each client will compute the forces exerted on by all the other atoms. The optimal number of atoms for client i is denoted as $n_i, 1 \leq i \leq p$, where p is the number of clients at the time the scheduler begins execution. The scheduling algorithm that we have developed computes the optimal $n_i, \forall i$ at every time step based on a continually and dynamically updated system model. The scheduling algorithm is implemented on the server side as part of this thread. It also performs the time integration of all the received forces. In practice, it is possible to parallelize the time integration, but we chose not to since it is of linear complexity and can be computed very fast in serial.

```
1: Read initial positions and velocities from file.
2: Spawn the server thread.
3: **for** each timestep **do**
4:     Call the scheduler to determine the optimal $n_i, 1 \leq$
       $i \leq p$.
5:     Signal to the client handlers that the position vector
       is ready.
6:     Wait for all the clients to return their forces.
7:     Perform time integration.
8: **end for**
```

Figure 3. Control thread of server

The server's server thread is responsible for spawning a new client handler thread every time a new client establishes a connection with the server. It is also responsible for keeping track of the available nodes and which nodes are taking part in the current time step's computation.

The server's client handler thread shown in Fig. 4, first benchmarks the client node's performance by running three test MD simulations. The size of these simulations is much smaller than the real simulation being performed. A wider separation in the number of atoms used in the benchmark allows our system model (Section 4) to perform more accurately. The execution times are noted both on the server side and on the client side, allowing us to compute two polynomials. The server side polynomial $f_i^s(x)$ allows us to model the computation and communication time, while the client side polynomial $f_i^c(x)$ allows us to model the computation time on the client node. Once the benchmark is completed, it informs the main control thread that client i is ready for computation and when it is assigned atoms to compute forces on, it first distributes all N atom positions and informs the client on which atoms it should compute forces. The execution time of the client node is again measured both from the server side and the client side and the previously generated polynomials are updated with this new data. We use the lagrange interpolation technique to generate these polynomials. When updating the polynomial, we replace the previously used point with the largest $x-$ co-ordinate before evaluating the lagrange interpolation formula since this is the point that dominates the behavior of the polynomial.

The scheduling algorithm is described in more detail in Section 5. The scheduler considers only clients that are available at the start of its execution. Clients that join in after the scheduler begins execution are not scheduled for the current time step, but are considered when scheduling for the next time step.

```
1:  Create three systems of atoms with $N_a$, $N_b$ and $N_c$ atoms each.
2:  Have client $i$ compute the forces on all the atoms for each of the systems of atoms generated.
3:  Let $t_a^s, t_b^s, t_c^s$ be the time taken to obtain the forces at the server side corresponding to the above simulations.
4:  Let $t_a^c, t_b^c, t_c^c$ be the execution times observed at the client side.
5:  Compute a second-degree polynomial $f_i^s(x)$ based on $(N_a, t_a^s), (N_b, t_b^s), (N_c, t_c^s)$ using lagrange interpolation.
6:  Compute a second-degree polynomial $f_i^c(x)$ based on $(N_a, t_a^c), (N_b, t_b^c), (N_c, t_c^c)$ using lagrange interpolation.
7:  **while** true **do**
8:      Wait for the control thread to signal that the position vector is ready.
9:      Send the positions of all $N$ atoms to client $i$.
10:     Let $t_x^s$ be the execution time of the clients force computation measured from the server side.
11:     Let $t_x^c$ be the execution time of the clients force computation measured from the client side.
12:     Update the polynomials $f_i^c(x)$ and $f_i^s(x)$ based on the points $(N, t_x^c \cdot N/n_i)$ and $(N, t_x^s \cdot N/n_i)$ respectively using lagrange interpolation.
13:     Signal that force computation by client $i$ is completed.
14: **end while**
```

Figure 4. Client handler thread of server for client i

Since the communication is always client initiated in this model, our framework works well in real-world settings since it is not hindered by firewalls and NATs on the client side. As long as a machine can establish outbound connections (which is the usual firewall policy), it can take part in the computation. The server is the only machine that has to be directly connected to the network and has a single port opened to allow incoming connections from the clients. This way, the clients can be submitted as jobs to condor pools, PBS queues, LSF queues etc., and as soon as they are scheduled to run, they will contact the server and begin computation. The server can dynamically handle the addition of nodes. In theory, it can also handle a node failure. However, deciding when a node has failed is a challenge. We have not yet implemented node failure detection in our code.

Our implementation is in C++ using sockets. Directly using Berkley sockets avoids having to depend on the client machine having libraries like PVM and MPI installed. Further, the communication patterns in this application are simple enough to implement directly in Berkley sockets. However, since threads are used, the actual implementation has to deal with synchronization issues and signals have to be emulated by conditional variables along with mutexes.

4 System Model

Since we are using a pair potential to model inter-atomic interactions, the time complexity for the molecular dynamics simulation is $\Theta(N^2)$. This allows the running time to be approximated by a second degree polynomial. In practice, we have observed that the running time can be extrapolated from such a polynomial within a tolerable relative error. For example, using the execution times of systems with 23328, 10976 and 5324 atoms, the execution time of a system with 131072 atoms can be predicted with a relative error of 2.2%.

The polynomial can be constructed for a given node by benchmarking it with simulations of sizes that are as widely separated as possible, such as $\{n, \frac{n}{2}, \frac{n}{4}\}$, and observing the corresponding running times $\{t_1, t_2, t_3\}$. Now, the polynomial coefficients can be computed using lagrange interpolation within a tolerable relative error.

If we have p nodes, we can construct the following model for the running times t_i^s's and t_i^c's, measured from the server side and client side respectively, in terms of the simulation size x_i as follows:

$$\begin{array}{ccccc}
t_1^s & = & f_1^s(x_1) & = & a_1^s x_1^2 + b_1^s x_1 + c_1^s, \\
t_2^s & = & f_2^s(x_2) & = & a_2^s x_2^2 + b_2^s x_2 + c_2^s, \\
\vdots & & \vdots & & \vdots \\
t_p^s & = & f_p^s(x_p) & = & a_p^s x_p^2 + b_p^s x_p + c_p^s, \\
t_1^c & = & f_1^c(x_1) & = & a_1^c x_1^2 + b_1^c x_1 + c_1^c, \\
t_2^c & = & f_2^c(x_2) & = & a_2^c x_2^2 + b_2^c x_2 + c_2^c, \\
\vdots & & \vdots & & \vdots \\
t_p^c & = & f_p^c(x_p) & = & a_p^c x_p^2 + b_p^c x_p + c_p^c.
\end{array}$$

where, $a_i^s, b_i^s, c_i^s, a_i^c, b_i^c and c_i^c$ are constants that are determined by an initial benchmark.

The fraction of time that a client node i will spend on communication can be predicted as

$$comm^{i}_{frac} = \frac{f_i^s(N) - f_i^c(N)}{f_i^s(N)} \tag{8}$$

This model also allows us to predict the mean fraction of time that will be spent on communication as:

$$comm_{frac} = \frac{\sum_{i=1}^{p} n_i \left[f_i^s(N) - f_i^c(N)\right]}{\sum_{i=1}^{p} n_i f_i^s(N)} \tag{9}$$

The client handler thread for client node i (Fig. 4) can additionally compute $comm^{i}_{frac}$ to determine whether it is above a specified threshold to ensure that we are not wasting too much time in communicating with the client node i instead of performing computation on it. If it is above the threshold, the client node can be rejected. Similarly, the expression $comm_{frac}$ can be evaluated and used to determine whether we should stop accepting new client nodes due to diminishing returns.

The communication bandwidth requirements are linear in the number of atoms. Hence, it will be modeled dominantly by the linear term in the server's polynomial. The startup times and network latencies are a constant overhead and will be modeled by the constant term in both the polynomials. The computational complexity is modeled dominantly by the quadratic term in the polynomials.

5 Scheduling Problem

An optimal schedule can be determined based on the system model described in section 4. If the current number of atoms being simulated is N, the execution time on node i is $t_i = f_i^s(N), 1 \leq i \leq p$. If we are computing the forces exerted only on n_i atoms, the execution time on node i from the server's point of view can be approximated as

$$t_i = \frac{n_i}{N} f_i^s(N),\ 1 \leq i \leq p \tag{10}$$

An optimal scheduler will determine the number of atoms to assign to each client node such that they all have the same execution time. The optimal execution time for all the nodes t_{opt}, can be determined as follows:

$$\begin{aligned} t_{opt} &= t_1 = t_2 = \ldots = t_p \\ &= \frac{n_1}{N} f_1^s(N) = \frac{n_2}{N} f_2^s(N) = \ldots \frac{n_p}{N} f_p^s(N) \end{aligned} \tag{11}$$

$$\text{subject to} \left\{ \begin{array}{rcl} \sum n_i &=& N, 1 \leq i \leq p \\ n_i &\in& \mathbb{Z}^+, \forall i \end{array} \right.$$

In the case of a simple pair potential, we have a set of linear equations modeling the execution times on the available nodes. In this case, it is simple to solve the entire model in closed form. However, the algorithm we present in section 6 is capable of finding an optimal assignment without taking into consideration the form of the equations used to model the execution times (i.e. it will work for arbitrary degree polynomials and even non-linear forms). This can be the case when using potentials that are more complex than the simple pair-potential that we use in this paper. The only requirement is that the equations used in the model must be strictly increasing functions (a reasonable assumption since the execution time grows with the problem size).

For a linear model, the closed form solution can be determined by simple algebraic manipulation. The optimal execution time and optimal assignment of atoms are determined by equations 12 and 13.

$$t_{opt} = \frac{1}{\sum \frac{1}{f_i^s(N)}} \tag{12}$$

$$n_i = \frac{N}{f_i^s(N) \sum \frac{1}{f_i^s(N)}} \tag{13}$$

We evaluate the accuracy of our scheduling algorithm presented in section 6 with the results obtained when using equations 12 and 13.

6 Scheduling Algorithm

It can be seen that the n_i values monotonically increase with t_i since the $f_i^s(N)$'s are positive and constant within each time step. This allows the optimal assignment of force computations to the clients to be determined using a binary search algorithm as shown in (Fig. 5). The scheduling algorithm makes use of the system model that we construct at the beginning of the simulation and update at every time step.

The algorithm works by picking an initial value for t and computing the number of atoms that can be assigned to each client node n_i such that all of them are busy computing the forces on the assigned atoms for exactly t units of time. Then, the sum $\sum_{i=1}^{p} n_i$ is computed which gives the total number of atoms on which forces can be computed in time t using all the available client nodes. If this sum is less than the number of atoms in the simulation N, we increase t until we can compute the forces on all N atoms using all the available p client nodes. Instead of searching linearly, we use a binary search to improve the time complexity of the scheduler. The algorithm terminates when N_{opt} is sufficiently close to N (determined by δ). After termination, if $N \neq N_{opt}$, the unassigned atoms if any (which can be at most δ) are assigned to the nodes in a uniform fashion.

From the pseudo code of Fig. 5, it can be seen that the algorithm executes in $O(p \cdot log\ t_U^{(0)})$ time. The initial

```
 1: $t_U \leftarrow t_U^{(0)}$
 2: $t_L \leftarrow 0$
 3: **while** $(t_L < t_U)$ **do**
 4:     $t \leftarrow (t_L + t_U)/2$
 5:     $N_{opt} \leftarrow \sum_{i=1}^{p}$ ROUND$(Nt/f_i^s(N))$
 6:     **if** $(|N_{opt} - N| < \delta)$ **then**
 7:         break
 8:     **else if** $(N_{opt} < N)$ **then**
 9:         $t_L \leftarrow t$
10:     **else**
11:         $t_U \leftarrow t$
12:     **end if**
13: **end while**
14: $t_{opt} \leftarrow t$
15: $n_i \leftarrow Nt_{opt}/f_i^s(N)$
16: **if** $(N_{opt} \neq N)$ **then**
17:     $\delta' \leftarrow |N_{opt} - N|$
18:     **if** $(\delta' > p)$ **then**
19:         **for** every node $i$ **do**
20:             $n_i \leftarrow n_i + \lfloor \frac{\delta'}{p} \rfloor$
21:         **end for**
22:         $\delta' \leftarrow \delta' - p\lfloor \frac{\delta'}{p} \rfloor$
23:     **end if**
24:     **for** the first $\delta'$ nodes $i$ **do**
25:         $n_i \leftarrow n_i + 1$
26:     **end for**
27: **end if**
```

Figure 5. Optimal Scheduling Algorithm

value $t_U^{(0)}$ can be approximated by using any of the polynomials $f_k^s(x)$ to determine the execution time if all nodes performed the same as node k i.e. $\frac{1}{p}f_k^s(N)$ and multiply this value by some sufficiently large constant λ. In practice, the value λ depends on how heterogeneous the underlying distributed system is. For most cases, a value between 5 and 10 seems to be sufficient. In all of our tests, we have seen convergence within 20 iterations of the binary search.

7 Potential Enhancements to the Scheduler

The performance of the scheduling algorithm described above can be improved using the techniques described in this section.

In a very dynamic environment, i.e. the available computational power on the client nodes varies significantly with time, instead of updating the polynomial coefficients with data from the previous time step, we can update the polynomials with a weighted average of the execution times over multiple steps in the past. The weights can either be determined experimentally or specified *a priori*. The reason for doing so is a large continuous variance in the available computational power on the client nodes can introduce significant oscillations in the relative error of our model.

From the pseudo code of the scheduler shown in Fig. 5, it can be noticed that the performance of the client nodes is sampled every time step. However, since the execution time of a time step is relatively long, a better schedule can be computed if the performance is sampled more frequently. This can be achieved by using a multi-round scheduling technique. In multi-round scheduling, instead of having the scheduler compute the forces on all N atoms, it can be modified to compute the forces on only N/r atoms within a round, where r is the number of rounds per time step. This allows for a finer granularity in the scheduling since we now sample the client nodes r times every time step instead of just once per time step as before. With multi-round scheduling, there is no significant increase in the total amount of data sent over the network. It is only the force vectors that are returned to the server by the client nodes that are sent in r chunks instead of as a single vector. The number of rounds should not be too large since this method is affected by network latencies. Thus, multi-round scheduling will allow us to get a more accurate performance model of the system.

Another technique that can be used, if the performance model is continually erring in its predictions for a given node, is to disallow it from taking part in the next time step and benchmarking it with additional data sets of varying sizes. This allows us to evaluate the lagrange interpolation formula on three fresh points. Since the time for execution of a time step is much larger than the time it takes to perform a benchmark, it makes more sense to use this technique when multi-round scheduling is used, since the node will be unavailable for computation only for the duration of a round.

8 Performance Evaluation

We use condor as the testbed for the proposed MD framework. Condor is a cycle scavenging batch scheduler. It seeks to utilize the idle time on nodes and regular desktops to run jobs that are submitted to its queue. It is highly configurable and supports advanced features such as job check pointing and migration. To run our MD application, we submit as many instances of the client program as required to condor. When condor finds nodes that are idle, it schedules the client jobs to run on those nodes.

To observe the impact of the heterogeneity of the distributed system on our application, we assign each client to compute the forces on exactly N/p atoms, where p is the number of clients nodes. Figure 6 illustrates the execution times on 5 client nodes over a period of 9 time steps of the MD simulation. It can be seen that the faster nodes such as node 4 and node 5 spend a lot of time waiting for the slower client nodes to complete execution.

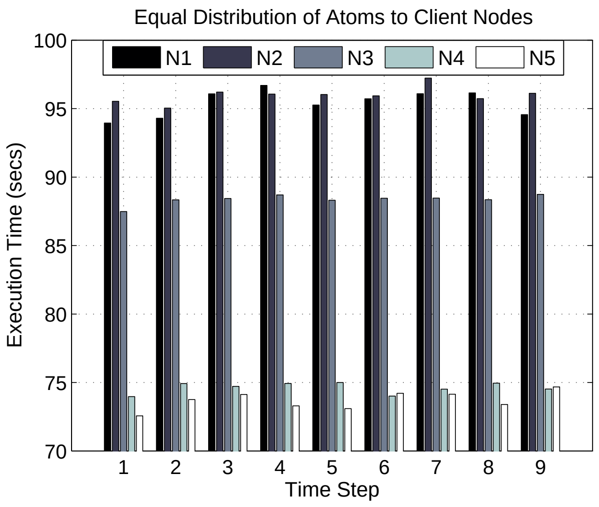

Figure 6. Simple scheduling: equally distributing the load among the nodes. Illustrates the differences in available compute power.

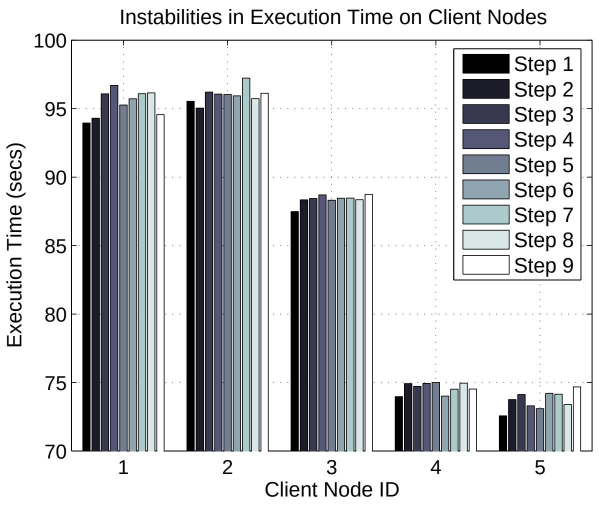

Figure 7. Execution of equal load multiple times. Illustrates the variations in available compute power.

To depict the variance in available computational power for a given client node, (Fig. 7) plots the execution times on each client node for multiple time steps with the same number of atoms (N/p) scheduled to the client nodes for all the time steps. A variation of about 4 seconds in the execution times can be observed in the client nodes. When repeated for a few hundred time steps, a variation of about 8% was observed. The reason for this is that the computational resources on the client nodes are shared with the node's owners background jobs such as email software, automatic operating system updates, anti-virus software, firewall software etc. These instabilities are of a lower order if the client jobs are submitted to a queueing system in a cluster environment such as PBS or LSF since nodes are dedicated to the jobs submitted to them.

Next, we use the scheduler developed in this paper to determine the optimal assignment of atoms to compute forces on the client nodes. Figure 8 depicts these atom assignments. It can be observed that the faster client nodes are assigned more atoms. Figure 9 illustrates our scheduling algorithm varying the atom assignments to client nodes based on the performance of the node in the previous time step. This is achieved by the recomputing of polynomial coefficients at every time step based on the execution time in the previous time step.

Figure 10 plots the execution time on the client nodes for 6 time steps. The aim of the scheduler is to use the system model to ensure that all client nodes complete execution at the same time. It does so by using the system model to predict the execution time for a particular assignment of atoms to client nodes. The efficiency of the scheduler depends on the accuracy of the system model. The horizontal lines in the plot depict the predicted execution times for all the client nodes. It can be seen that most of the client nodes finish execution very close to these lines and the ones that do not, get closer to the predicted value in the next time step, since the polynomials in the system model were updated based on the execution time in the previous time step. Since the polynomial based on server measured time is used in scheduling, it can track changes in the network as well as in the available computational power of the client nodes.

As mentioned earlier, our scheduling algorithm can handle new client nodes becoming available arbitrarily. Scheduling decisions are made at the start of every time step. New client nodes that become available during the execution of a time step are assigned atoms to compute forces during the next time step. Figure 11 plots the adaptation to a new client node being added to the simulation at time step 3. Since the new client node is already benchmarked at the time the scheduler executes, it can be optimally scheduled even for the first time step of the computation it takes part in.

Figure 12 plots the relative error of the predictions made by our scheduler over a period of 100 time steps. When computing the relative error, we do not use the absolute value function in the numerator, since the sign of the error can tell us whether real execution time was less than or greater than the predicted execution time. A positive relative error indicates that the real execution time exceeded the predicted execution time and vice versa. Client node 6 is added to the computation only at time step 36. From

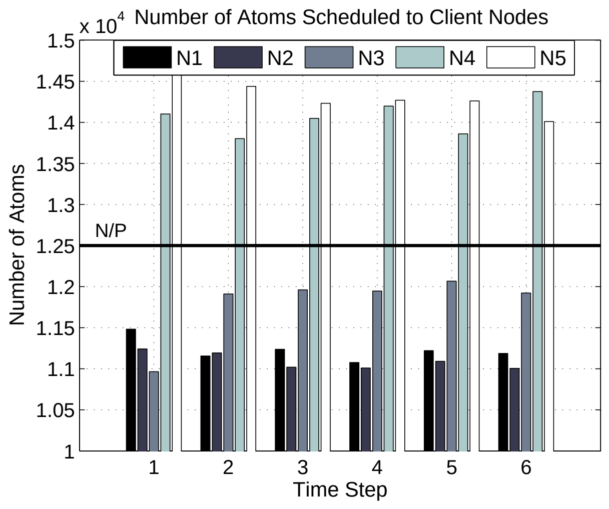

Figure 8. The proposed scheduling algorithm adjusting the number of atoms sent to each node based on its performance during the previous time step.

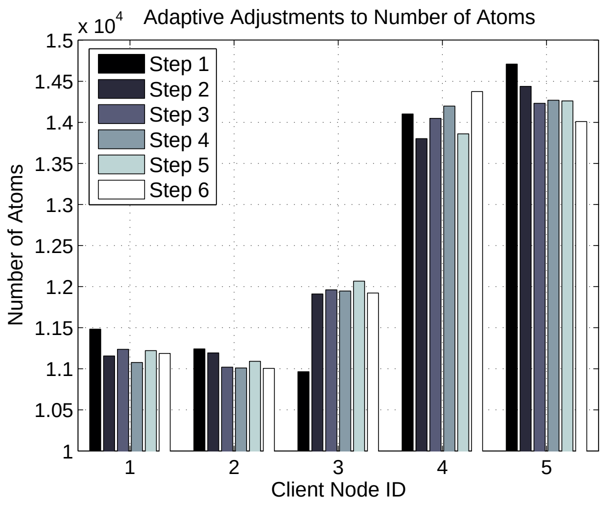

Figure 9. Atom assignments using the proposed scheduling algorithm. More atoms are sent to the faster nodes. The simple N/P method on the other hand assigns 12500 atoms to all the nodes.

the plots, it can be observed that adaptive changes to the polynomial coefficients allows the changes in the available computational power to be compensated for within the next few time steps.

If a client node takes longer to execute than predicted (due to some transient load), it is assigned fewer atoms to compute forces on in the next time step due to the adaptivity in our scheduling algorithm. However, if the previous transient load disappears during the next time step, the real execution time will be less than the predicted value. This is the reason for the alternating positive and negative relative errors in the plot. The magnitude of these oscillations can be reduced by using some of the techniques described in section 7.

All the MD simulations performed in the above experiments involved systems of 62500 atoms initialized to a FCC lattice. The client nodes on which the simulations were scheduled have different CPU speeds and memory capacities. Three of the nodes were connected to via a 100Mbps network and the other via a 10Mbps network. Random network traffic was generated during the experiments by running a bittorrent [14] client and having it download a Linux ISO file.

9 Future Work

There exist many interesting aspects of the proposed scheduling technique that can be investigated further. The Grid is a very dynamic environment in which new nodes can appear at anytime and existing nodes may arbitrarily go down. Our algorithm currently handles the case of new nodes being added to the computation dynamically. It can deal with nodes arbitrarily going down if it can efficiently detect that a node is no longer available for the computation. We are currently investigating various techniques to deal with detecting and handling client node failure. Some of the issues with this problem involve how long to wait before deciding that a node is dead and the optimal way to deal with the failure. Further, we are also looking into how to determine the optimal number of rounds to use when using the multi-round scheduling discussed in section 7. We are also looking into combining our approach with other scheduling techniques such as divisible load theory and control theory. Our goal is to create a generalized framework that can be used for other scientific applications including those whose execution time cannot be well approximated by a n^{th} degree polynomial.

The developed scheduling algorithm is not specific only to MD. It can be used with any parallel application where the execution time can be approximated by an n^{th} degree polynomial. There are numerous applications that fall under this category such as most matrix operations on dense matrices, image processing algorithms, numerical algorithms for solving partial differential equations, etc. Many applications whose execution time depends only on the size of the input data set and not on the nature of the elements in the input data set fall under this category.

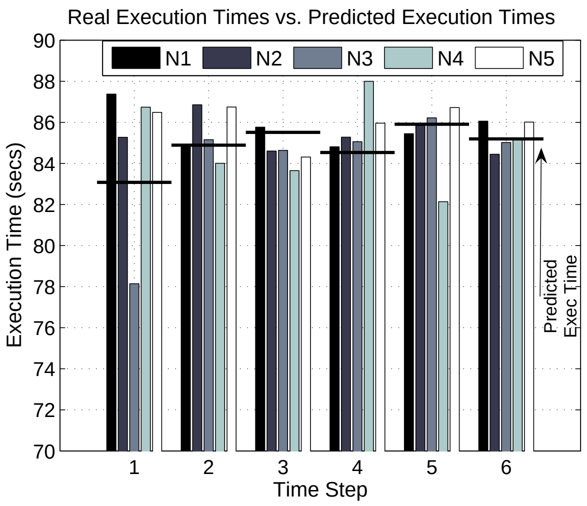

Figure 10. Our scheduling algorithm: Execution time on each node for six time steps. Horizontal line depicts the finish time predicted by our model.

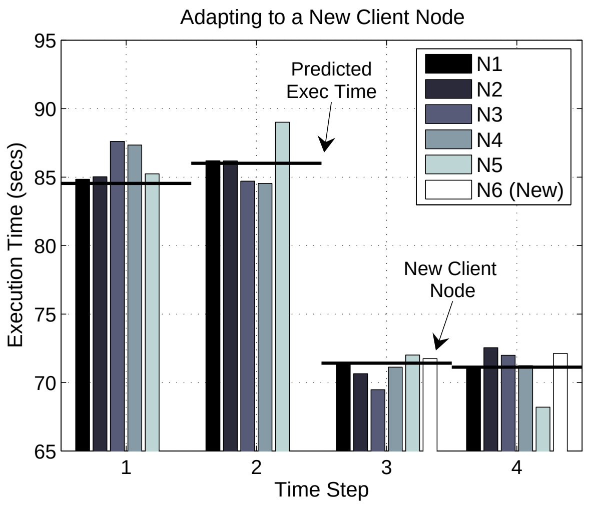

Figure 11. Our scheduling algorithm: A new node is added at time step 3. The model automatically predicts its finish time and rescales atoms to the other nodes.

10 Conclusion

Computational molecular dynamics is a very computationally intensive application. Fortunately, it can be parallelized. However, when scheduling in a heterogeneous distributed environment such as the Grid, the continually changing available network latency, bandwidth and computational power with respect to each client node must be taken into consideration. We present an efficient scheduling algorithm that is capable of dynamically load balancing long range MD. The algorithm is capable of adapting to changes in the performance of the client nodes at runtime by continually examining their past performance and varying the distributed load accordingly. Our algorithm can be used for other similar applications and will work as long as the execution times can be modeled by monotonically increasing functions. We have also evaluated the performance of the scheduling algorithm in a condor pool with heterogeneous networking and available compute power.

References

[1] Efficiency of dynamic load balancing based on permanent cells for parallel molecular dynamics simulation. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 85, Washington, DC, USA, 2000. IEEE Computer Society.

[2] M. P. Allen and D. J. Tildesley. *Computer simulation of liquids*. Oxford University Press, 1987.

[3] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Comput.*, 36(5):570–580, 1987.

[4] S. Chapman and T. G. Cowling. *The Mathematical Theory of Non-uniform Gases*. Cambridge University Press, 1970.

[5] T. W. Clark, R. v. Hanxleden, J. A. McCammon, and L. R. Scott. Parallelizing molecular dynamics using spatial decomposition. In *Proceedings of the Scalable High–Performance Computing Conference*, pages 95–102. IEEE Computer Soc. Press, 1994.

[6] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150:1–??, 2001.

[7] D. F. Hegarty and M. T. Kechadi. Topology preserving dynamic load balancing for parallel molecular simulations. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–19, New York, NY, USA, 1997. ACM Press.

[8] C. F. C. J. O. Hirschfelder and R. B. Bird. *Molecular Theory of Gases and Liquids*. Wiley, 1954.

[9] S. Krishnan and L. V. Kale. A parallel adaptive fast multipole algorithm for n-body problems. In *Proceedings of the International Conference on Parallel Processing*, pages III 46 – III 50, August 1995.

[10] M. Litzkow and M. Livny. Experience with the Condor distributed batch system. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, Huntsville, AL, October 1990.

[11] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. *IBM, Ottawa, Canada*, 1966.

[12] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics, Sandia Report, SAND91-1144 (1993)., 1993.

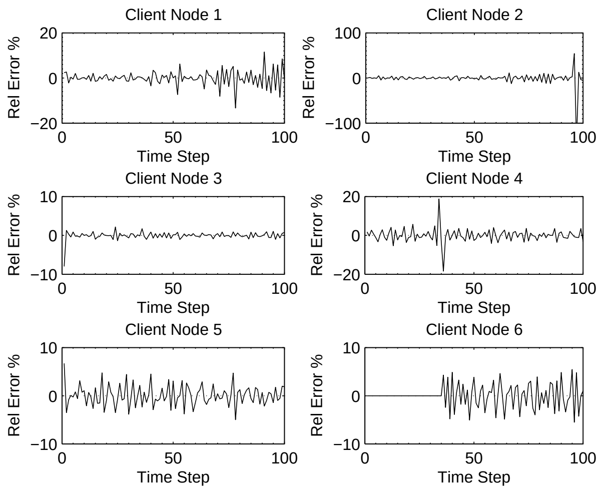

Figure 12. Relative errors in the predictions of our model. A positive relative error indicates that the node took longer to execute than predicted. Node 6 is added to the computation only at time step 36.

[13] S. Plimpton and B. Hendrickson. A new parallel method for molecular dynamics simulation of macromolecular systems. Sandia Technical Report, SAN94-1862, 1994., 1994.

[14] D. Qiu and R. Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 367–378, New York, NY, USA, 2004. ACM Press.

[15] S.Gupta. Computing aspects of molecular dynamics simulations. In *J.Comp.Phys.Comm.*, volume 70:243-270, 1992.

[16] J. V. Sumanth, D. R. Swanson, and H. Jiang. Scheduling many-body short range md simulations on a cluster of workstations and custom vlsi hardware. In L. Bougé and V. K. Prasanna, editors, *HiPC*, volume 3296 of *Lecture Notes in Computer Science*, pages 166–175. Springer, 2004.

[17] L. Verlet. Computer experiments on classical fluids i. thermodynamical properties of lennard-jones molecules. In *Phys. Rev.*, volume 159:98-103, 1967.

[18] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Supercomputing*, pages 12–21, 1993.