NAMD Programming Guide

Version 1.5

R. Brunner, A. Dalke, A. Gursoy, W. Humphrey, M. Nelson

September 4, 1998

Theoretical Biophysics Group
University of Illinois and Beckman Institute
405 N. Mathews
Urbana, IL 61801

Description

The program NAMD is a parallel, object-oriented molecular dynamics program designed for efficiency, portability, and modularity. A general structure based on a small set of generic objects for communication between processors, calculation of atomic forces calculation, and integration of the equations of motion allows NAMD to adopt different algorithms for these aspects of the molecular dynamics calculation.

The program uses a spatial decomposition to distribute atoms among processors. The model is broken down into uniform cubes of space referred to as patches. A set of patches are assigned to each processor. Patches are dynamically redistributed during the simulation to achieve better load-balancing. The program is message driven, meaning that the order of computation is determined by the arrival of messages indicating that the data necessary for a given computation is available. This type of scheduling of tasks allows for the greatest possible overlap of communication and computation times. NAMD uses the same force field and input files as the program X-PLOR. It will also produce binary DCD trajectory files and will be able to interact with the graphical display program VMD.

The NAMD *Programming Guide* provides a complete description of what NAMD does and how it does it. This guide describes the design and implementation of NAMD, providing a researcher with the ability to understand the program without examining the source code.

NAMD Version 1.5

Authors: Robert Brunner, Andrew Dalke, Attila Gursoy,
Bill Humphrey, and Mark Nelson

Theoretical Biophysics Group, Beckman Institute, University of Illinois.

NOTICE

The program NAMD is *not* in the public domain. However, it is freely available without fee for education, research, and commercial purposes. By obtaining copies of this and other files that comprise the NAMD program, you, the Licensee, agree to abide by the following conditions and understandings with respect to the copyrighted software:

1. The software is copyrighted in the name of the Board of Trustees of the University of Illinois (UI), and ownership of the software remains with the UI.

2. Permission to use and modify this software and its documentation is hereby granted to Licensee. In addition, permission to copy this work and any derived works is granted to Licensee, provided that

   (a) the copyright notice and this permission notice appear on all such copies,

   (b) that proper credit be given by citing

      M. Nelson, W. Humphrey, A. Gursoy, A. Dalke, L. Kale, R. Skeel and K. Schulten. NAMD - A parallel, object-oriented molecular dynamics program. *J. Supercomputing App.*, 10:251-268, 1996.

   for NAMD and

      W. T. Rankin and J. A. Board, Jr., Tech Rept. 95-002, EE Dept., Duke Univ.

   for DPMTA

   (c) that NAMD and any derived works are not redistributed for a fee,

   (d) that programs derived from NAMD be given a different name.

3. Licensee may not use the name, logo, or any other symbol of the UI nor the names of any of its employees nor any adaptation thereof in advertizing or publicity pertaining to the software without specific prior written approval of the UI.

4. THE UI MAKES NO REPRESENTATIONS ABOUT THE SUITABILITY OF THE SOFTWARE FOR ANY PURPOSE. IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR IMPLIED WARRANTY.

5. The UI shall not be liable for any damages suffered by Licensee from the use of this software.

Contents

List of Figures

1 Introduction

This document is the Programmer's Guide for the parallel molecular dynamics program NAMD. It is intended to provide a detailed guide to exactly what NAMD does, how it does it, and why it does it in this way. This manual is divided in several distinct sections. Section 2 details exactly what NAMD computes. This includes details of and derivations for the force field that is computed and the integration scheme used. Section 3 details the design of NAMD. This section details the overall design of the program, including the details of the multithreaded, message driven design and the spatial decomposition scheme used to accomplish it. Section 4 describes the basics of working with NAMD, including how to set up a working directory for the program, the coding conventions used, and the details of the configuration file used to run NAMD. The last section of the guide is the longest and most detailed. This section provides the details of the implemenation of NAMD, including a detailed description of every class used in the program.

The goal of this document is to provided the reader enough detail about the design and implementation of NAMD that they will be able to modify the program to implement new algorithms and methods.

2 Computational Specification

This section details exactly what NAMD calculates during a simulation. The first section describes the various components of the force field that is used, the second section describes the integration method used, and the last section describes the units and constant values used by the program.

2.1 Force Field Details

The force fields used by NAMD are compatible with those used by the programs X-PLOR and CHARMm. It includes energy terms for electrostatics, van der Waals, linear bonds, angular bonds, dihedral bonds, improper bonds, and Hydrogen bonds. The total energy function used by NAMD is:

$$E_{total} = E_{elec} + E_{vdw} + E_{bond} + E_{angle} + E_{d/i} + E_{constraint}$$

The following subsections describe the individual energy terms in detail as well as how to determine the forces associated with each type of interaction.

2.1.1 Electrostatics

Electrostatics describes the force resulting from the interaction between two charged particles. The electrostatic energy between two atoms i and j is described by Coulomb's Law as:

$$E_{elec} = \epsilon_{14} \frac{C q_i q_j}{\epsilon_0 |\vec{r}_{ij}|}$$

where

ϵ_{14} = Scaling factor for 1-4 interactions. This is zero for 1-2 and 1-3 interactions and is equal to 1.0 for any interaction other than a modified 1-4 interaction. The value used for modified 1-4 interactions is specified in the configuration file using the `1-4scaling` parameter. (See section 4.3)

$C = 2.31 \times 10^{-19}$ J nm

q_i = charge of atom i from .psf file

q_j = charge of atom j from .psf file

ϵ_0 = dielectric constant specified in the configuration file using the parameter `dielectric`. (See section 4.3)

$\vec{r}_{ij} = \vec{r}_j - \vec{r}_i$, the vector from atom i to atom j

$|\vec{r}_{ij}|$ = length of $\vec{r}_{ij}$, which is equal to the calculated distance between atoms i and j

By differentiating this formula, the force exerted by this electrostatic interaction is:

$$\vec{F}_{elec} = \epsilon_{14} \frac{C q_i q_j}{\epsilon_0 |\vec{r}_{ij}|^2} \hat{r}_{ij}$$

where

$\hat{r}_{ij} = \frac{\vec{r}_{ij}}{|\vec{r}_{ij}|}$, a unit vector in the direction of $\vec{r}_{ij}$

2.1.2 Van der Waals

The Van der Waals interactions describe the forces resulting from local interactions of atoms. The Van der Waals energy between two atoms i and j is described by:

$$E_{vdw} = \frac{A}{|\vec{r}_{ij}|^{12}} - \frac{B}{|\vec{r}_{ij}|^{6}}$$

where

A = constant determined as described below

B = constant determined as described below

$|\vec{r}_{ij}|$ = calculated distance between atoms i and j

The constants A and B can be specified for a pair of atom types explicitly in the parameter file using the `NBFix` command. With this command, values of A and B for normal interactions and modified 1-4 interactions are specified explicitly.

If the `NBFix` command is not used, the constants A and B are calculated using the parameters σ_{ij} and ϵ_{ij} using the equations:

$$A = 4{\sigma_{ij}}^{12}\epsilon_{ij}$$
$$B = 4{\sigma_{ij}}^{6}\epsilon_{ij}$$

$\sigma_i j$ and $\epsilon_i j$ are calcuated from the σ and ϵ values specifed for the atom types of atoms i and j in the `NBOnd` commands in the parameter file using the equations:

$$\sigma_{ij} = \frac{\sigma_{ii} + \sigma_{jj}}{2}$$
$$\epsilon_{ij} = \sqrt{\epsilon_{ii}\epsilon_{jj}}$$

where

σ_{ii} = σ value for atom i

σ_{jj} = σ value for atom j

Also, σ and ϵ for a pair can be related to the minimum energy, E_{min}, and minimum distance, R_{min} by the equations:

$$R_{min} = \sigma\sqrt[6]{2}$$
$$E_{min} = -\epsilon$$

For modified 1-4 interactions, the σ_{ij} and ϵ_{ij} values are calculated as above, except that the σ_{ii}, ϵ_{ii}, σ_{jj}, and ϵ_{jj} are replaced by the σ^{14}_{ii}, ϵ^{14}_{ii}, σ^{14}_{jj}, and ϵ^{14}_{jj} that are specified by the `NBOnd` lines of the parameter files. Section 2.1.3 describes when the modified parameters are used.

The force for the Van der Waals interactions obtained by differentiating the energy function above is:

$$\vec{F}_{vdw} = \left(\frac{12A}{{r_{ij}}^{13}} - \frac{6B}{{r_{ij}}^{7}}\right)\hat{r}_{ij}$$

where

$\hat{r}_{ij} = \frac{\vec{r}_{ij}}{|\vec{r}_{ij}|}$, a unit vector in the direction of $\vec{r}_{ij}$

2.1.3 Bonded Exclusions

Certain pairs of atoms are excluded from electrostatic and Van der Waals calculations because of their bonded interactions. The rules to apply in choosing bonded exclusions are specifed in the configuration file using the `exclude` parameter. The choices for exclusions are `none`, `1-2`, `1-3`, `1-4`, and `scaled1-4`. With `none`, no atom pairs are excluded. With `1-2`, only atoms that are connected via a linear bond are excluded. With `1-3`, any pair of atoms connected via a bond or bonded to a common third atom are excluded. With `1-4`, any atoms that are connected by a linear bond, or a sequence of two bonds, or a sequence of three bonds are excluded. With `scaled1-4`, exlcusions are applied in the same manner as the `1-3` setting, but those pairs that are connected by a sequence of 3 bonds are calculated using the modified 1-4 methods described rather than the standard force calculations.

2.1.4 Switching functions

NAMD allows the cutting off of the electrostatic and Van der Waals forces to be handled in two ways. The default way is to simply truncate the forces at the cutoff value specified in the configuration file using the parameter `cutoff`. In this case, if the distance r between two atoms is less than the cutoff, then the forces are calculated as specified above. If r is greater than the cutoff, then the forces and energies are simply set to 0. But this method leads to a discontinuity in the force field. As atoms move within the cutoff distance, there electrostatic and Van der Waals energies suddenly jump from 0 to some finite value.

The other means of dealing with these cutoff provided by NAMD is switching functions. These functions smoothly bring the forces and energies to 0 at the cutoff distance to avoid any discontinuities in the force field. These switching functions are equivalent to those used in X-PLOR when the `VSWitch` and `SHIft` options are specified. They are activated using the `switching` parameter in the configuration file.

There are different switching function used for electrostatics and Van der Waals. For electrostatics, the energy function is modified to be:

$$E_{elec} = \epsilon_{14} \frac{C q_i q_j}{\epsilon_0 |\vec{r}_{ij}|} \left(1 - \frac{|\vec{r}_{ij}|^2}{{R_{off}}^2}\right)^2 \text{ where } |\vec{r}_{ij}| < R_{off}$$

$$E_{elec} = 0 \text{ where } |\vec{r}_{ij}| \geq R_{off}$$

For van der Waals interactions, the energy function is modified to be:

$$E_{vdw} = \left(\frac{A}{|\vec{r}_{ij}|^{12}} - \frac{B}{|\vec{r}_{ij}|^6}\right) SW(|\vec{r}_{ij}|)$$

where SW is the switching function defined as:

$$SW(r_{ij}) = 0 \text{ if } |\vec{r}_{ij}| > R_{off}$$

$$SW(r_{ij}) = 1 \text{ if } |\vec{r}_{ij}| \leq R_{on}$$

$$SW(r_{ij}) = \frac{\left({R_{off}}^2 - {r_{ij}}^2\right)^2 \left({R_{off}}^2 + 2{r_{ij}}^2 - 3{R_{on}}^2\right)}{\left({R_{off}}^2 - {R_{on}}^2\right)^3} \text{ if } R_{off} > |\vec{r}_{ij}| \geq R_{on}$$

where

R_{on} is a constant defined using the configuration value `switchdist`

R_{off} is specfied using the configuration value `cutoff`

Since the energy functions are modified by multiplying by a function of r, the forces applied are also affected. For the electrostatic force, the switched force is:

$F_{elec} = \epsilon_{14} \frac{Cq_iq_j}{\epsilon_0 |\vec{r}_{ij}|^2} \left(1 - \frac{|\vec{r}_{ij}|^2}{{R_{off}}^2}\right)^2 + \left(\epsilon_{14} \frac{Cq_iq_j}{\epsilon_0 |\vec{r}_{ij}|}\right) 4 \left(1 - \frac{|\vec{r}_{ij}|^2}{{R_{off}}^2}\right) \frac{|\vec{r}_{ij}|}{{R_{off}}^2}$ where $|\vec{r}_{ij}| < R_{off}$

$F_{elec} = 0$ where $|\vec{r}_{ij}| \geq R_{off}$

The Van der Waals forces are modified to be:

$F_{vdw} = \left(\frac{12A}{|\vec{r}_{ij}|^{13}} - \frac{6B}{|\vec{r}_{ij}|^7}\right)$ if $|\vec{r}_{ij}| \leq R_{on}$

$F_{vdw} = 0$ if $|\vec{r}_{ij}| > R_{off}$

$$\begin{aligned} F_{vdw} &= \left(\frac{12A}{|\vec{r}_{ij}|^{13}} - \frac{6B}{|\vec{r}_{ij}|^7}\right) SW(|\vec{r}_{ij}|) \\ &- \left(\frac{A}{|\vec{r}_{ij}|^{12}} - \frac{B}{|\vec{r}_{ij}|^6}\right) 4| \\ &\times \vec{r}_{ij}| \left(\left({R_{off}}^2 - |\vec{r}_{ij}|^2\right)^2 - \left({R_{off}}^2 - |\vec{r}_{ij}|^2\right)\left({R_{off}}^2 + 2|\vec{r}_{ij}|^2 - 3{R_{on}}^2\right)\right) \end{aligned}$$

if $R_{off} \geq |\vec{r}_{ij}| > R_{on}$

2.1.5 Bonds

Bonds describe a linear bond between two atoms. These bonds are described by simple harmonic springs. The energy of a bond between atoms i and j is given by:

$$E_{bond} = k(|\vec{r}_{ij}| - r_0)^2$$

where

$k =$ spring constant specified in the parameter file for this bond type

$|\vec{r}_{ij}| =$ calculated distance between atoms i and j

$r_0 =$ rest distance of the bond specified in the parameter file for this bond type

By differentiating this formula, the force for this bond can be found to be:

$$\vec{F}_{bond} = -2k(|\vec{r}_{ij}| - r_0)\hat{r}_{ij}$$

where

$\hat{r}_{ij} = \frac{\vec{r}_{ij}}{|\vec{r}_{ij}|}$, a unit vector in the direction of $\vec{r}_{ij}$

2.1.6 Angles

Angles describe angular bonds between three atoms. These bonds are modeled as harmonic angular springs. The energy of such a bond between atoms i, j, and k is given by:

$$E_{angle} = k_\theta(\theta - \theta_0)^2 + k_{ub}(|\vec{r}_{ik}| - r_{ub})^2$$

where

k_θ = force constant specified in the parameter file for this bond type

θ = calculated angle between the vector that connect atoms i and j and the vector that connects atoms k and j. This angle is calculated using the formula:

$$\theta = \cos^{-1}\left(\frac{\vec{r}_{ij} \cdot \vec{r}_{kj}}{|\vec{r}_{ij}||\vec{r}_{kj}|}\right)$$

θ_0 = rest angle for this bond specified in the parameter file for this angle type

k_{ub} = Urey-Bradley constant, which defaults to zero

r_{ub} = rest distance for the Urey-Bradley term

In order to determine the forces acting on the three atoms involved in the angular bond, the gradient of the energy function must be determined. The derivation begins with the following:

$$\vec{F}_{angle} = -\nabla\left(k_\theta(\theta - \theta_0)^2 + k_{ub}(|\vec{r}_{ik}| - r_{ub})^2\right),$$

$$\vec{F}_{angle} = -2k_\theta(\theta - \theta_0)\nabla\theta(\vec{r}_i, \vec{r}_j, \vec{r}_k) - 2k_{ub}(|\vec{r}_{ik}| - r_{ub})\nabla|\vec{r}_{ik}|,$$

with

$$\theta(\vec{r}_i, \vec{r}_j, \vec{r}_k) = \cos^{-1}\left(\frac{\vec{r}_{ij} \cdot \vec{r}_{kj}}{|\vec{r}_{ij}||\vec{r}_{kj}|}\right) = \cos^{-1}(\cos(\theta)).$$

Using these relations, it is left to reader to prove that the forces on atoms i, j, and k can be found using the equations:

$$F_x^i = \left(\frac{2k_\theta(\theta - \theta_0)}{\sin(\theta)|\vec{r}_{ij}|}\right)\left(\frac{x_{ij}\cos(\theta)}{|\vec{r}_{ij}|} - \frac{x_{kj}}{|\vec{r}_{kj}|}\right) - 2k_{ub}(|\vec{r}_{ik}| - r_{ub})\frac{x_{ik}}{|\vec{r}_{ik}|}$$

$$F_y^i = \left(\frac{2k_\theta(\theta - \theta_0)}{\sin(\theta)|\vec{r}_{ij}|}\right)\left(\frac{y_{ij}\cos(\theta)}{|\vec{r}_{ij}|} - \frac{y_{kj}}{|\vec{r}_{kj}|}\right) - 2k_{ub}(|\vec{r}_{ik}| - r_{ub})\frac{y_{ik}}{|\vec{r}_{ik}|}$$

$$F_z^i = \left(\frac{2k_\theta(\theta - \theta_0)}{\sin(\theta)|\vec{r}_{ij}|}\right)\left(\frac{z_{ij}\cos(\theta)}{|\vec{r}_{ij}|} - \frac{z_{kj}}{|\vec{r}_{kj}|}\right) - 2k_{ub}(|\vec{r}_{ik}| - r_{ub})\frac{z_{ik}}{|\vec{r}_{ik}|}$$

$$F_x^k = \left(\frac{2k_\theta(\theta - \theta_0)}{\sin(\theta)|\vec{r}_{kj}|}\right)\left(\frac{x_{kj}\cos(\theta)}{|\vec{r}_{kj}|} - \frac{x_{ij}}{|\vec{r}_{ij}|}\right) - 2k_{ub}(|\vec{r}_{ik}| - r_{ub})\frac{x_{ki}}{|\vec{r}_{ik}|}$$

$$F_y^k = \left(\frac{2k_\theta(\theta - \theta_0)}{\sin(\theta)|\vec{r}_{kj}|}\right)\left(\frac{y_{kj}\cos(\theta)}{|\vec{r}_{kj}|} - \frac{y_{ij}}{|\vec{r}_{ij}|}\right) - 2k_{ub}(|\vec{r}_{ik}| - r_{ub})\frac{y_{ki}}{|\vec{r}_{ik}|}$$

$$F_z^k = \left(\frac{2k_\theta(\theta-\theta_0)}{\sin(\theta)|\vec{r}_{kj}|}\right)\left(\frac{z_{kj}\cos(\theta)}{|\vec{r}_{kj}|} - \frac{z_{ij}}{|\vec{r}_{ij}|}\right) - 2k_{ub}(|\vec{r}_{ik}| - r_{ub})\frac{z_{ki}}{|\vec{r}_{ik}|}$$

$$F_x^j = -(F_x^i + F_x^k)$$

$$F_y^j = -(F_y^i + F_y^k)$$

$$F_z^j = -(F_z^i + F_z^k)$$

2.1.7 Dihedrals and Impropers

Dihedral and improper bonds model the interaction between 4 bonded atoms. They are modeled by an angular spring between the planes formed by the first 3 atoms and the second 3 atoms. The energy for a dihedral or improper between atoms i, j, k, and l is given by:

$$E_{d/i} = k(1+\cos(n\phi+\delta))$$

if n>0 or

$$E_{d/i} = k(\phi-\delta)^2$$

if n=0. where

k = force constant specified in the parameter file for this bond type

ϕ = calculated angle between the plane formed by atoms i, j, and k and the plane formed by atoms j, k, and l

n = periodicity of the bond specified in the parameter file for this bond type

δ = phase shift specified in the parameter file for this bond type

The angle ϕ is calculated by first determining the vectors $\vec{A}$ and $\vec{B}$ where:

$$\vec{A} = \vec{r}_{ij} \times \vec{r}_{jk}$$

$$\vec{B} = \vec{r}_{jk} \times \vec{r}_{kl}$$

These two vectors can then be used to calculate ϕ using the formula

$$\phi = \cos^{-1}\left(\frac{\vec{A}\cdot\vec{B}}{|\vec{A}||\vec{B}|}\right)$$

To determine the force, the negative gradient of the energy must be found. It can be shown that if $n=0$ then the force is given by

$$\vec{F} = (2k(\phi-\delta))(\nabla\phi)$$

and if $n>0$ then the force is given by

$$\vec{F} = (nk\sin(n\phi+\delta))(\nabla\phi)$$

Using the formula for ϕ given, it can be shown that:

$$\nabla\phi = \left(\frac{1}{\sin(\phi)}\right)\nabla\left(\frac{\vec{A}\cdot\vec{B}}{|\vec{A}||\vec{B}|}\right)$$

But this can lead to a singularity if $\sin(\phi)$ goes to 0. Therefore following the method used by X-PLOR, if $\sin(\phi)$ is nearly 0, then a third vector $\vec{C}$ is determined using

$$\vec{C} = \vec{r}_{jk} \times \vec{A}$$

If the angle ψ is the angle between $\vec{C}$ and $\vec{B}$ then it can be shown that

$$\cos(\phi) = \sin(\psi)$$

It can then be shown that

$$\phi = \sin^{-1}\left(\frac{\vec{C}\cdot\vec{B}}{|\vec{C}||\vec{B}|}\right) + \frac{\pi}{2}$$

and therefore

$$\nabla\phi = \left(\frac{1}{\cos(\phi)}\right)\nabla\left(\frac{\vec{C}\cdot\vec{B}}{|\vec{C}||\vec{B}|}\right)$$

By then expressing $\vec{A}$, $\vec{B}$, and $\vec{C}$ in terms of the corrdinates of i, j, k, and l, it is left to the reader to prove that for the first formulation, the force can be found to be:

$$F_x^i = K\frac{1}{\sin(\phi)}\frac{1}{|\vec{A}|}\left(r_y^{jk}(\hat{B}_z - \cos(\phi)\hat{A}_z) - r_z^{jk}(\hat{B}_y - \cos(\phi)\hat{A}_y)\right)$$

$$F_y^i = K\frac{1}{\sin(\phi)}\frac{1}{|\vec{A}|}\left(r_z^{jk}(\hat{B}_x - \cos(\phi)\hat{A}_x) - r_x^{jk}(\hat{B}_z - \cos(\phi)\hat{A}_z)\right)$$

$$F_z^i = K\frac{1}{\sin(\phi)}\frac{1}{|\vec{A}|}\left(r_x^{jk}(\hat{B}_y - \cos(\phi)\hat{A}_y) - r_y^{jk}(\hat{B}_x - \cos(\phi)\hat{A}_x)\right)$$

$$\begin{aligned} F_x^j = \quad & K \quad \frac{1}{\sin(\phi)}\left(\frac{1}{|\vec{A}|}\left[r_z^{ij}(\hat{B}_y - \cos(\phi)\hat{A}_y) - r_y^{ij}(\hat{B}_z - \cos(\phi)\hat{A}_z)\right]\right. \\ + \quad & \left.\frac{1}{|\vec{B}|}\left[r_y^{kl}(\hat{A}_z - \cos(\phi)\hat{B}_z) - r_z^{jk}(\hat{A}_y - \cos(\phi)\hat{B}_y)\right]\right) - F_x^i \end{aligned}$$

$$\begin{aligned} F_y^j = \quad & K \quad \frac{1}{\sin(\phi)}\left(\frac{1}{|\vec{A}|}\left[r_x^{ij}(\hat{B}_z - \cos(\phi)\hat{A}_z) - r_z^{ij}(\hat{B}_x - \cos(\phi)\hat{A}_x)\right]\right. \\ + \quad & \left.\frac{1}{|\vec{B}|}\left[r_z^{kl}(\hat{A}_x - \cos(\phi)\hat{B}_x) - r_x^{jk}(\hat{A}_z - \cos(\phi)\hat{B}_z)\right]\right) - F_y^i \end{aligned}$$

$$\begin{aligned} F_z^j = \quad & K \quad \frac{1}{\sin(\phi)}\left(\frac{1}{|\vec{A}|}\left[r_y^{ij}(\hat{B}_x - \cos(\phi)\hat{A}_x) - r_x^{ij}(\hat{B}_y - \cos(\phi)\hat{A}_y)\right]\right. \\ + \quad & \left.\frac{1}{|\vec{B}|}\left[r_x^{kl}(\hat{A}_y - \cos(\phi)\hat{B}_y) - r_y^{jk}(\hat{A}_x - \cos(\phi)\hat{B}_x)\right]\right) - F_z^i \end{aligned}$$

$$F_x^k = -F_x^l - \quad K \quad \frac{1}{\sin(\phi)} \left(\frac{1}{|\vec{A}|} \left[r_z^{ij} (\hat{B}_y - \cos(\phi)\hat{A}_y) - r_y^{ij} (\hat{B}_z - \cos(\phi)\hat{A}_z) \right] \right.$$
$$\left. + \quad \frac{1}{|\vec{B}|} \left[r_y^{kl} (\hat{A}_z - \cos(\phi)\hat{B}_z) - r_z^{jk} (\hat{A}_y - \cos(\phi)\hat{B}_y) \right] \right)$$

$$F_y^k = -F_y^l \quad K \quad \frac{1}{\sin(\phi)} \left(\frac{1}{|\vec{A}|} \left[r_x^{ij} (\hat{B}_z - \cos(\phi)\hat{A}_z) - r_z^{ij} (\hat{B}_x - \cos(\phi)\hat{A}_x) \right] \right.$$
$$\left. + \quad \frac{1}{|\vec{B}|} \left[r_z^{kl} (\hat{A}_x - \cos(\phi)\hat{B}_x) - r_x^{jk} (\hat{A}_z - \cos(\phi)\hat{B}_z) \right] \right)$$

$$F_z^k = -F_z^l - \quad K \quad \frac{1}{\sin(\phi)} \left(\frac{1}{|\vec{A}|} \left[r_y^{ij} (\hat{B}_x - \cos(\phi)\hat{A}_x) - r_x^{ij} (\hat{B}_y - \cos(\phi)\hat{A}_y) \right] \right.$$
$$\left. + \quad \frac{1}{|\vec{B}|} \left[r_x^{kl} (\hat{A}_y - \cos(\phi)\hat{B}_y) - r_y^{jk} (\hat{A}_x - \cos(\phi)\hat{B}_x) \right] \right)$$

$$F_x^l = K \frac{1}{\sin(\phi)} \frac{1}{|\vec{B}|} \left(r_z^{jk} (\hat{B}_y - \cos(\phi)\hat{A}_y) - r_y^{jk} (\hat{B}_z - \cos(\phi)\hat{A}_z) \right)$$
$$F_y^l = K \frac{1}{\sin(\phi)} \frac{1}{|\vec{B}|} \left(r_x^{jk} (\hat{B}_z - \cos(\phi)\hat{A}_z) - r_z^{jk} (\hat{B}_x - \cos(\phi)\hat{A}_x) \right)$$
$$F_z^l = K \frac{1}{\sin(\phi)} \frac{1}{|\vec{B}|} \left(r_y^{jk} (\hat{B}_x - \cos(\phi)\hat{A}_x) - r_x^{jk} (\hat{B}_y - \cos(\phi)\hat{A}_y) \right)$$

where

$$K = 2k(\phi - \delta)$$

if $n = 0$ and

$$K = nk\sin(n\phi + \delta)$$

if $n > 0$.
and that for the second formulation, the components of the force are:

$$F_x^i = K \frac{1}{\cos(\phi)} \frac{1}{|\vec{C}|} \left(\left({r_y^{jk}}^2 + {r_z^{jk}}^2 \right) (\hat{B}_x - \sin(\phi)\hat{C}_x) - r_x^{jk} r_y^{jk} (\hat{B}_y - \sin(\phi)\hat{C}_y) - r_x^{jk} r_z^{jk} (\hat{B}_z - \sin(\phi)\hat{C}_z) \right)$$

$$F_y^i = K \frac{1}{\cos(\phi)} \frac{1}{|\vec{C}|} \left(\left({r_z^{jk}}^2 + {r_x^{jk}}^2 \right) (\hat{B}_y - \sin(\phi)\hat{C}_y) - r_y^{jk} r_z^{jk} (\hat{B}_z - \sin(\phi)\hat{C}_z) - r_y^{jk} r_x^{jk} (\hat{B}_x - \sin(\phi)\hat{C}_x) \right)$$

$$F_z^i = K \frac{1}{\cos(\phi)} \frac{1}{|\vec{C}|} \left(\left({r_x^{jk}}^2 + {r_y^{jk}}^2 \right) (\hat{B}_z - \sin(\phi)\hat{C}_z) - r_z^{jk} r_x^{jk} (\hat{B}_x - \sin(\phi)\hat{C}_x) - r_z^{jk} r_y^{jk} (\hat{B}_y - \sin(\phi)\hat{C}_y) \right)$$

$$\begin{aligned}
F_x^j = & \quad K \quad \frac{1}{\cos(\phi)}\left(\frac{1}{|\vec{C}|}\Big(-(r_y^{jk}r_z^{ij}+r_z^{jk}r_z^{ij})(\hat{B}_x-\sin(\phi)\hat{C}_x)+(2r_x^{jk}r_y^{ij}-r_y^{jk}r_x^{ij})(\hat{B}_y-\sin(\phi)\hat{C}_y)\right. \\
& + \quad \left.(2r_x^{jk}r_z^{ij}-r_z^{jk}r_x^{ij})(\hat{B}_z-\sin(\phi)\hat{C}_z)\Big)+\frac{1}{|\vec{B}|}\Big(r_y^{kl}(\hat{C}_z-\sin(\phi)\hat{B}_z)-r_z^{kl}(\hat{C}_x-\sin(\phi)\hat{B}_x)\Big)\right)-F_x^i
\end{aligned}$$

$$\begin{aligned}
F_y^j = & \quad K \quad \frac{1}{\cos(\phi)}\left(\frac{1}{|\vec{C}|}\Big(-(r_z^{jk}r_z^{ij}+r_x^{jk}r_z^{ij})(\hat{B}_y-\sin(\phi)\hat{C}_y)+(2r_y^{jk}r_z^{ij}-r_z^{jk}r_y^{ij})(\hat{B}_z-\sin(\phi)\hat{C}_z)\right. \\
& + \quad \left.(2r_y^{jk}r_x^{ij}-r_x^{jk}r_y^{ij})(\hat{B}_x-\sin(\phi)\hat{C}_x)\Big)+\frac{1}{|\vec{B}|}\Big(r_z^{kl}(\hat{C}_x-\sin(\phi)\hat{B}_x)-r_x^{kl}(\hat{C}_z-\sin(\phi)\hat{B}_z)\Big)\right)-F_y^i
\end{aligned}$$

$$\begin{aligned}
F_z^j = & \quad K \quad \frac{1}{\cos(\phi)}\left(\frac{1}{|\vec{C}|}\Big(-(r_x^{jk}r_x^{ij}+r_y^{jk}r_y^{ij})(\hat{B}_z-\sin(\phi)\hat{C}_z)+(2r_z^{jk}r_x^{ij}-r_z^{jk}r_x^{ij})(\hat{B}_x-\sin(\phi)\hat{C}_x)\right. \\
& + \quad \left.(2r_z^{jk}r_y^{ij}-r_y^{jk}r_z^{ij})(\hat{B}_y-\sin(\phi)\hat{C}_y)\Big)+\frac{1}{|\vec{B}|}\Big(r_x^{kl}(\hat{C}_y-\sin(\phi)\hat{B}_y)-r_y^{kl}(\hat{C}_x-\sin(\phi)\hat{B}_x)\Big)\right)-F_z^i
\end{aligned}$$

$$\begin{aligned}
F_x^k = - & \quad K \quad \frac{1}{\cos(\phi)}\left(\frac{1}{|\vec{C}|}\Big(-(r_y^{jk}r_z^{ij}+r_z^{jk}r_z^{ij})(\hat{B}_x-\sin(\phi)\hat{C}_x)+(2r_x^{jk}r_y^{ij}-r_y^{jk}r_x^{ij})(\hat{B}_y-\sin(\phi)\hat{C}_y)\right. \\
& + \quad \left.(2r_x^{jk}r_z^{ij}-r_z^{jk}r_x^{ij})(\hat{B}_z-\sin(\phi)\hat{C}_z)\Big)+\frac{1}{|\vec{B}|}\Big(r_y^{kl}(\hat{C}_z-\sin(\phi)\hat{B}_z)-r_z^{kl}(\hat{C}_x-\sin(\phi)\hat{B}_x)\Big)\right)-F_x^l
\end{aligned}$$

$$\begin{aligned}
F_y^k = - & \quad K \quad \frac{1}{\cos(\phi)}\left(\frac{1}{|\vec{C}|}\Big(-(r_z^{jk}r_z^{ij}+r_x^{jk}r_z^{ij})(\hat{B}_y-\sin(\phi)\hat{C}_y)+(2r_y^{jk}r_z^{ij}-r_z^{jk}r_y^{ij})(\hat{B}_z-\sin(\phi)\hat{C}_z)\right. \\
& + \quad \left.(2r_y^{jk}r_x^{ij}-r_x^{jk}r_y^{ij})(\hat{B}_x-\sin(\phi)\hat{C}_x)\Big)+\frac{1}{|\vec{B}|}\Big(r_z^{kl}(\hat{C}_x-\sin(\phi)\hat{B}_x)-r_x^{kl}(\hat{C}_z-\sin(\phi)\hat{B}_z)\Big)\right)-F_y^l
\end{aligned}$$

$$\begin{aligned}
F_z^k = - & \quad K \quad \frac{1}{\cos(\phi)}\left(\frac{1}{|\vec{C}|}\Big(-(r_x^{jk}r_x^{ij}+r_y^{jk}r_y^{ij})(\hat{B}_z-\sin(\phi)\hat{C}_z)+(2r_z^{jk}r_x^{ij}-r_z^{jk}r_x^{ij})(\hat{B}_x-\sin(\phi)\hat{C}_x)\right. \\
& + \quad (2r_z^{jk}r_y^{ij}-r_y^{jk}r_z^{ij})(\hat{B}_y-\sin(\phi)\hat{C}_y)\Big) \\
& + \quad \left.\frac{1}{|\vec{B}|}\Big(r_x^{kl}(\hat{C}_y-\sin(\phi)\hat{B}_y)-r_y^{kl}(\hat{C}_x-\sin(\phi)\hat{B}_x)\Big)\right)-F_z^l
\end{aligned}$$

$$F_x^l = -K\frac{1}{\cos(\phi)}\frac{1}{|\vec{B}|}\Big(r_z^{jk}(\hat{C}_y-\sin(\phi)\hat{B}_y)-r_y^{jk}(\hat{C}_z-\sin(\phi)\hat{B}_z)\Big)$$

$$F_y^l = -K\frac{1}{\cos(\phi)}\frac{1}{|\vec{B}|}\Big(r_x^{jk}(\hat{C}_z-\sin(\phi)\hat{B}_z)-r_z^{jk}(\hat{C}_x-\sin(\phi)\hat{B}_x)\Big)$$

$$F_z^l = -K \frac{1}{\cos(\phi)} \frac{1}{|\vec{B}|} \left(r_y^{jk} (\hat{C}_x - \sin(\phi) \hat{B}_x) - r_x^{jk} (\hat{C}_y - \sin(\phi) \hat{B}_y) \right)$$

where K is again

$$K = 2k(\phi - \delta)$$

if $n = 0$ and

$$K = nk \sin(n\phi + \delta)$$

if $n > 0$.

2.1.8 Harmonic Constraints

Harmonic constrants provide a mechanism for holding certain parts of a molecule relatively immobile during a simulation. It allows specific atoms to be held to a reference position. The energy for an atom i that is constrained is:

$$E_{constraint} = k_i (|\vec{r}_i - \vec{r}_{ref}|)^e$$

where

k_i = Force constant defined for atom i

$\vec{r}_i$ = Current position of atom i

$\vec{r}_{ref}$ = Reference position atom i is constrained to

e = Exponent defined for all constraints

The force applied by the harmonic constraints is:

$$\vec{F}_{constraint} = ek_i (|\vec{r}_i - \vec{r}_{ref}|)^{e-1} \hat{r}_{i-ref}$$

where

$\hat{r}_{i-ref}$ = Unit vector in the direction from $\vec{r}_i$ to $\vec{r}_{ref}$

The reference positions, force constaints, exponent, and atoms which are constrained are defined by the user using several configuration options. For more details, see the description of the options `constraints`, `consexp`, `conskfile`, `conskcol`, and `consref` in section 4.3.

Steered Molecular Dynamics in NAMD NAMD provides a feature called “moving constraints” that allows one to run SMD simulations. This feature is is based on the harmonic constraints feature, and effectively allows for the reference position of a specified atom to move with a constant velocity $\vec{v}$. The potential for this atom is computed according to

$$U_{SMD}(t) \;=\; k \left[\vec{r}(t) \;-\; \vec{r}_i(t)\right]^e ,$$

and the force acting on atom i is (by differentiating U_{SMD})

$$\vec{f}_{SMD}(t) \;=\; -\, ek \left[\vec{r}(t) \;-\; \vec{r}_i(t)\right] ,$$

where t is time, $\vec{r}_i$ is the position of the atom i and $\vec{r}$ is the current reference position (position of the restraint point), given by

$$\vec{r}(t) \; = \; \vec{r}_0 \, + \, \vec{v}t \, .$$

In addition to the usual parameters used for harmonic constraints, one should specify the number (0-based indexing) of the constrained atom (`movingConsAtom`), the reference position of which (specified in the usual manner) will move according to Eq. 2.1.8. Also a velocity vector $\vec{v}$ (`movingConsVel`) needs to be specified.

The way the moving constraints work is that the moving reference position is calculated every integration time step using Eq. 2.1.8, where $\vec{v}$ is in Å/timestep, and t is the current timestep (i.e., `firstTimestep` plus however many timesteps have passed since the beginning of NAMD run). Therefore, one should be careful when restarting simulations to appropriately update the `firstTimestep` parameter in the NAMD configuration file or the reference position specified in the reference PDB file.

NOTE: NAMD actually calculates the constraints potential with $U = k(x - x_0)^d$ and the force with $F = dk(x - x_0)$, where d is the exponent `consexp`. The result is that if one specifies some value for the force constant k in the PDB file, effectively, the force constant is $2k$ in calculations.

2.1.9 Spherical Boundary Conditions

NAMD provides spherical boundary conditions that consist of a single or two superimposed potential functions. The potential energy for these energies is given by:

$$E_{sphere} = k_{sphere}(|\vec{r}_i - \vec{r}_{center}| - r_{sphere})^{exp_{sphere}}$$

if $|\vec{r}_i - \vec{r}_{center}| > r_0$ and $E_{sphere} = 0$ otherwise.

k_{sphere} = force constant

$\vec{r}_i$ = Current position of atom i

$\vec{r}_{center}$ = Center of the spherical boundary

r_{sphere} = Raidus of the spherical boundary

exp_{sphere} = Exponent for the potential function

The force applied by this potential is given by:

$$\vec{F}_{sphere} = \left(exp_{sphere} k_{sphere} (|\vec{r}_i - \vec{r}_{center}| - r_{sphere})^{exp_{sphere}-1} \right) \hat{r}_{i,center}$$

where

$\hat{r}_{i,center}$ = a unit vector in the direction from atom i to the center of the sphere.

Thus a positive force constant will cause a force that moves atoms back in to the center of the sphere and a negative force constant will force atoms away from the sphere.

For simple harmonic boundary conditions, a single potential with an exponent of 2 or 4 may be used to obtain a potential like that shown in figure 1.

It may be desirable, however, to simulate a surface tension affect, where a small harmonic well around the sphere boundary is desired. To obtain such an effect, two potentials of the form

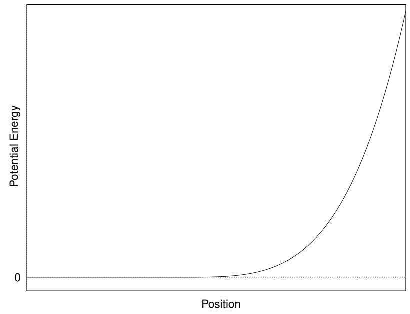

Figure 1: Potential energy function for a spherical boundary condition using a single potential.

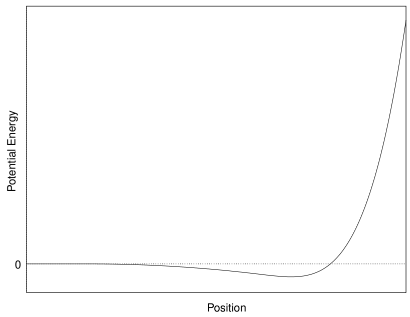

Figure 2: Potential energy function for a spherical boundary condition using two potentials, one a negative quadratic and another a positive quartic, to approximate surface tension.

shown above can be superimposed. By combining a potential with a negative force constant and an exponent of two with a potential with a positive force constant and a exponent of four and a slightly larger raidus, a potential such as that shown in figure 2 can be obtained.

Currently, the spheres must be centered around the initial center of mass of the system. The force constant, radius, and exponents for each potential are defined in the configuration file using the options `sphericalBCk1`, `sphericalBCr1`, `sphericalBCexp1`, `sphericalBCk2`, `sphericalBCr2`, and `sphericalBCexp2`. These parameters are described in section 4.3.

2.2 Integration

NAMD provides several forms of integration for the force field described in the previous section. These methods are describing the following sections include: Verlet or leapfrog is used for normal dynamics; a form of steepest decent is provided for energy minimization; Langevin dynamics; and temperature rescaling.

2.2.1 Verlet

For normal dynamics simulation, NAMD uses the velocity form of the Verlet or Leapfrog method for integration. Begining at a timestep n and given the position, velocity, and force acting on each atom, X_n, V_n, and F_n, the following equations are used to obtain values for the next step:

$$V_{n+\frac{1}{2}} = V_n + \frac{\Delta t}{2} M^{-1} F_n$$
$$X_{n+1} = X_n + \Delta t V_{n+\frac{1}{2}}$$
$$F_{n+1} = F(X_{n+1})$$
$$V_{n+1} = V_{n+\frac{1}{2}} + \frac{\Delta t}{2} M^{-1} F_{n+1}$$

While this is the most natural way to state the method, this is not actually the order things are performed in NAMD. Instead, a normal timestep will look something like: Given X_n, $V_{n-\frac{1}{2}}$, and F_n:

$$V_n = V_{n-\frac{1}{2}} + \frac{\Delta t}{2} M^{-1} F_n$$

which is then used to calculate the kinetic energy at step n. Then

$$V_{n+\frac{1}{2}} = V_n + \frac{\Delta t}{2} M^{-1} F_n$$
$$X_{n+1} = X_n + \Delta t V_{n+\frac{1}{2}}$$

n is then incremented, and another force evaluation is done.

In order to keep the interface with the rest of the program more natural, the velocities at the half timestep interval will be stored internally in the `Integrate` object, and the velocities used by the rest of the program will be the velocities at each timestep boundary.

There are slightly different formulations of the Verlet algorithm that rely on positions X_n and X_{n-1}, but it is felt such formulations suffer from a higher degree of round-off error than the velocity formulation.

2.2.2 Minimization

NAMD provides a simple form of a depth first search to perform energy minimization. This method uses a modified form of the Verlet method described above. It is modified in two ways. The first is that the velocity of the particles is set to zero after each timestep. This means that direction of movement of each particle during an integration step will always be in the direction of the gradient, thus createing a depth first serach. The second modification is to place a bound on the movement of an atom during any timestep, since a structure with very high potential energies may experience very rapid movement of atoms during the beginning of minimization. The amount of movement per timestep can be specified using the configuration option `maximumMove`. If no value is specified, NAMD will assume a default value of

$$\frac{0.75 \times \texttt{pairlistDist}}{\texttt{stepsPerCycle}},$$

where the parameter `pairlistDist` is the bounding distance between pairs for inclusion in pairlists, and the parameter `stepsPerCycle` indicates the number of time steps between pairlist generation. See section 4.3 for more details about these configuration parameters. The default value for `maximumMove` insures that no atom can move more than 3/4 of the pairlist distance during a cycle. If greater accuracy in the energy calculation is desired, then this value should be reduced.

2.2.3 Langevin Dynamics

Simple Langevin dynamics are also provided by NAMD. This consists of adding a random force and subtracting a friction force from each atom during each integration step.

The random force is calculated such that the average force is zero and the standard deviation is:

$$2k_b T_0 b_i m_i \Delta t$$

where

k_b = Boltzman's constant

T_0 = Target temperature specified using the configuration parameter `langTemp`

b_i = Friction coefficient for atom i

m_i = Mass of atom i

Δt = Timestep size

The friction force applied is:

$$m_i b_i \frac{dx_i}{dt}$$

where

m_i = Mass of atom i

b_i = Friction coefficient for atom i

In order to apply these forces, NAMD uses the same third-order finite difference approximation in dt that X-PLOR uses. This approximation uses:

$$x_i{}^{n+1} = \left(1 + \frac{b_i \Delta t}{2}\right)\left(2x_i{}^n - x_i{}^{n-1} + F_i{}^n \frac{\Delta t^2}{m_i} + x_i{}^{n-1}\left(\frac{b_i \Delta t}{2}\right)\right)$$

to update the position of particle i from step n to step $n+1$ and:

$$v_n = \left(\frac{1}{2\Delta t}\right)(x_{n+1} - x_{n-1})$$

to update the velocity from step $n-1$ to step n.

2.2.4 Velocity Rescaling

NAMD allows the user to equilibrate a system to a specified temperature by periodically rescaling the velocities of all particles such that a specified temperature is achieved.

This rescaling is achieved by first applying the integration scheme to obtain a set of velocities for timestep n. These velocities are then rescaled to the desired temperature by multiplying by the factor:

$$\sqrt{\frac{T_0}{T_{ave}}}$$

where

$T_0 =$ the desired temperature specified using the configuration option `rescaleTemp`

$T_{ave} =$ the average temperature since the last velocity rescaling

The positions for timestep $n+1$ are then obtained by using these new velocities.

Temperature rescaling may be used with either Verlet integrator or with Langevin dynamics. The number of timesteps between rescalings is specified using the configuration parameter `rescaleFreq`.

2.2.5 Rigid Bonds

Rigid water is an option (rigid bond angle and bond lengths). Also bonds between hydrogens and heavy atoms can be constrained to their nominal length during integration. Three options for selecting what bonds to constrain are given in NAMD. The configuration parameter `rigidBonds` can be set to `none`, which indicates no constrained lengths, `water` to constrain only bonds in water molecules, or `all` to constrain all bonds between hydrogens and heavy atoms. Bonds between heavy atoms cannot currently be constrained, and NAMD will not work correctly (even with rigid bonds inactive) if the system contains hydrogen molecules, since no heavy (or *parent*) atom can be found for either hydrogen. The rigid bond code also cannot handle methane molecules, or any other molecule with four or more hydrogens bonded to a heavy atom.

The rigid bonds module sorts the atoms of interest into one of four data structures, according to how many hydrogen atoms are bonded to the parent atom. After applying the normal integration procedure, the atom positions are adjusted to bring the bond lengths back to their neutral length (ie. the length such that the bond exerts no force on either atom). These adjustments are applied in such a way that the center of mass of the bonded atoms does not move. For the one-hydrogen case, this is a straightforward calculation, but for the two- or three-hydrogen cases, an iterative procedure is followed, where each hydrogen-parent atom pair is adjusted until all the hydrogen bonds are within a certain tolerance of the desired length, or a maximum number of iterations is exceeded. These values are set by the `rigidTolerance` and `rigidIterations` parameters. A modified algorithm is applied to waters. Although waters have only two hydrogens, a constraint is also applied to the distance between the two hydrogens and the three-bond procedure followed, so that the angle between the hydrogens is also constrained.

After correcting the atom positions, the calculated velocities must also be adjusted to account for the changed positions. Without velocity correction, the kinetic energy is computed incorrectly. Velocity correction is performed by the Rattle module. The velocities calculated by the normal integration are adjusted by removing the component parallel to the bond, without changing the velocity of the center of mass of the bonded pair.

Constraints:

$$\chi^\alpha(X) = 0, \alpha = 1, 2, ..., \text{whatever}.$$

Algorithm: given $X_n, V_{n-\frac{1}{2}}$

crunch numbers: $F_n =$ forces
half-kick: $\bar{V}_n = V_{n-\frac{1}{2}} + \frac{1}{2}\Delta t M^{-1} F_n$
half-kick: $V_{n+\frac{1}{2}} = \bar{V}_n + \frac{1}{2}\Delta t M^{-1} F_n$
drift: $\bar{X}_{n+1} = X_n + \Delta t V_{n+\frac{1}{2}}$
SHAKE: $X_{n+1} = \bar{X}_{n+1} + \frac{1}{2}\Delta t^2 M^{-1} \sum_\beta \lambda_\beta \chi_x^\beta(X_n)$
where $\{\lambda_\beta\}$ solve

$\{\chi_\alpha(X_{n+1}) = 0\}$
$\{V_{n+\frac{1}{2}} = (X_{n+1} - X_n)/\Delta t\}$
RATTLE: Compute the new velocities using: $V_n = (I - M^{-1}\chi_x^{\mathrm{t}}(\chi_x M^{-1}\chi_x^{\mathrm{t}})^{-1}\chi_x)\bar{V}_n$

Note: The optional RATTLE step is for the purpose of computing kinetic energy. It forces velocities to satisfy velocity constraints, but these adjustments are exactly undone by the next SHAKE.

Rigid bond calculation can be performed with Verlet integration, with or without minimization or velocity rescaling. Although NAMD will run with Langevin integration and rigid bonds both active, the equilibrium temperature of the system does not converge to the correct value. Different Langevin equations are needed to produce the correct velocity distribution, which have not yet been added to NAMD.

2.3 Units and Constant Values

This section defines the units, constant values, and conversion factors that are used by NAMD.

Units

The units currently used by NAMD are:

Measure	Unit	Symbol
Length	Angstroms	Å
Time	picoseconds	ps
Energy	Kilocalories per Mole	Kcal/mol
Mass	Atomic Mass Unit	AMU
Charge	Electron charge	e
Temperature	degrees Kelvin	K

Constants

The constant values used by NAMD are:

Symbol	Value	Name
K_b	1.987191×10^{-3} KCal/(mol · K)	Boltzman's Constant
N_a	6.022045×10^{23} 1/mol	Avogadro's Number
C	332.0636 KCal · Å/(mol · e^2)	Coulomb's constant
e	1.6021892 Coulombs	Electron charge
π	3.1415926535898	Pi

2.4 Conversion Factors

The conversion factors currently used by NAMD are:

1 Kcal = 4184.0 Joule

1 AMU = $1.6605655 \times 10^{-27}$ Kilograms

1 Å= 1.0×10^{-10} Meters

3 Design

NAMD uses a multithreaded, message driven design build on top of a spatial decomposition of the molcular system to provide high performance, scalable parallel performance. The computationally intensive aspects of MD simulations is the evaluation of electrostatic forces due to all the particles in the system. NAMD employs a multiple time stepping algorithm to reduce the cost of such calculations. This section contains a description of all aspects of this design and why it was used.

3.1 NAMD is a message passing program

NAMD is a parallel message passing program. It is composed of p processes where p where p is the number of processors. If the underlying machine supports multiple processes per processor, p can be greater than the number of processors. The processes are numbered between 0 and $p-1$. Process 0 is referred as the master process. It has responsibilities beyond those of normal processes for such things as startup, shutdown, and input/output.

Communication between processes is accomplished using `Communicate`, a communication layer object, that resides in each process. This object provides and interface between the other objects in NAMD and the actual message passing system being used. Therefore, NAMD can be ported to any message passing system by modifying the `Communicate` class.

3.2 Multiple time-stepping

The multiple time-stepping algorithm is used to reduce the computational cost of computing electrostatic forces due to all pairs of particles in the system. In this algorithm, the total electrostatic force acting on each atom is divided into two parts, a short-range (local) component and a long-range component. The short-range component is the forces due to pairs that are separated by less than local interaction length — *electrostatic cutoff*. The long-range component consists of the forces due to remaining pairs, i.e. interactions outside of the local interaction length. Since the longe-range forces are varying slowly, they can be evaluated less frequently. NAMD evaluates them every k time-steps, where each set of k time-steps referred as a *cycle*. These long-range forces, then, reused in time-steps withing the cycle without recalculation. NAMD ignores the van der Waals forces beyond the local interaction distance. Therefore, the van der Waals cutoff distance forms a lower limit to the local interaction distance.

3.3 Spatial Decomposition

The parallelization strategy of NAMD is based of spatial decomposition. The space occupied by the simulated model is divided into cubes of space called *patches*. Each patch has dimensions which are the electrostatic interaction distance plus a small safety margin. These values are provided by the user in the simulation parameters: `pairlist-distance`, and `margin` parameters. The decomposition of the simulation domain into patches are done in the `PatchDistrib` object.

Each patch is responsible for the atoms within its region. This means that, each patch stores the current position, and velocity of each of these atoms, as well as gathering and computing all of the forces necessary to perform integration during each step. In order to gather and calculate the necessary forces, each patch need to communicate with its neighboring patches. In three dimension, a patch has 26 immediate neighbors. In this version, interacting beyond immediate neighbors are now allowed. That's why the patch size has a lower bound as the local interaction length.

3.4 Computations and Communication per timestep

In order to gather and calculate the necessary forces, each patch will have to communicate with its neighboring patches. This communication will take place in cycles. The computations and communications during a cycle can be divided into three groups: a) begin-cycle phase (during the first time step of the cycle), b) cycle phase (k time steps), and c) end-cycle phase (at the end of the k^{th} step in the cycle.

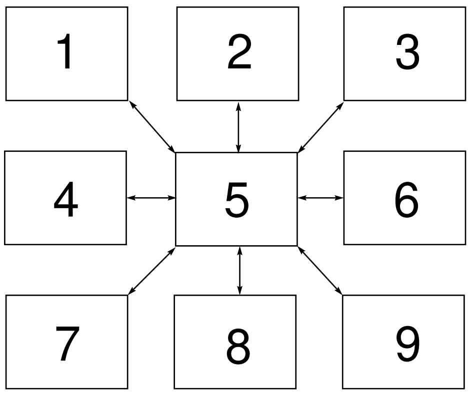

Figure 3: Diagram of patch communication

At the beginning of a cycle, first, each patch determines the interactions that it has with each neighboring patch, and sets up necessary data structure to be used within the cycle. For example, if a neighbor doesn't have any atoms, then there is no need to interact with that neighbor during the mid-cycle steps. The `PatchDistrib` class maintaines a list of neighboring patches for each patch, and also assigning the communication direction between each pair of patches. Therefore, at the beginning of each cycle, each patch receives two lists of neighbors from the `PatchDistrib` object. One list contains the neighboring patches that it will receive all atom coordinates from and the other will contain the neighboring patches that it will send all atom coordinates to. From the patches that the patch sends all of its atom coordinates to, it will receive coordinates for bonded interactions from. The determination of the direction of the communication between patches is part of the load balancing algorithm which is described in section 3.6 Second, if the full electrostatic (for example, DPMTA) option is on, then the patch initiates the long-range force calculation. The details of the interface to a full electrostatic module and calculation of long-range electrostatic forces are discussed in section 3.8 with more detail.

Each patch is responsible for calculating all short-range electrostatic, Van der Waals, and bonded interactions. These are the calculations performed during each time step within the cycle. Some of these interactions are completely local to the patch, i.e., atoms involved in these interactions owned by the patch. These interactions therefore don't require any communication of coordinates. The interactions that involve atoms from neighboring patches require the coordinates. This step involves communication with neighboring patches. The short-range electrostatic and the 2-atom bonded interactions are computed in one of the patches and the result is sent back to the other one (using Newton's Third Law). Calculation of the 3 and 4-body bonded interactions (angle, dihedral, and impropers), on the other hand, are duplicated in both neigboring patches than the use of Newton's Third Law to reduce the computation. Due to the small number of these interactions and the sizeable complexity in dealing with all the possible combinations of atoms, bonds, and patches (i.e., consider a dihedral bond with each of the four atoms residing in different patches ...) it is thought that this duplicate calculation does not result in significant overhead. At the end of a cycle, the list of atoms that each patch has is adjusted since atoms might move to a space occupied by a different patch. This phase is called atom redistribution.

As an example of these ideas, consider the two dimensional patches shown in Figure 3. Patch 5 would send all of its atom coordinates to some of its neighbors, say Patches 2, 3, 6, and 9. It would receive all of the coordinates from Patches 1, 4, 7, and 8. Based on the atoms received from Patches 1, 4, 7, and 8, Patch 5 would calculate the atoms that it needed to send to these patches so that they could compute their bonded interactions, and send these coordinates every timestep. So during each timestep, Patch 5 would send out atom coordinates to all 8 neighbors. To Patches 2, 3, 6, and 9 it would send all of its atom coordinates. To Patches 1, 4, 7, and 8, it would only send those coordinates necessary for bonded interactions. Patch 5 would expect to receive atom coordinates from all 8 neighbors. From Patches 2, 3, 6, and 9 it would only receive the coordinates necessary for bonded interactions. These coordinates would be used for force calculations and then discarded. No force messages would be sent back to these patches. From Patches 1, 4, 7, and 8, Patch 5 would receive all atom coordinates. Bonded, electrostatic, and Van der Waals forces would be calculated, and the electrostatic and Van der Waals forces would be communicated back to these patches. Patch 5 would also expect to receive a force message containing electrostatic and Van der Waals forces from Patches 2, 3, 6, and 9.

3.5 Multiple Patches per Processor and Message-Driven Design

One important decision in NAMD is to have more patches than the processes, and a message-driven style execution to achieve high performance. Each patch is implemented as an object that acts as its own thread of control. Each patch maintains its own state and contains functions which alter this state. Patches can send messages to each other by using their unique identifiers. When a patch receives a message, it responds by invoking appropriate functions determined form the content and the tag of the message. Each patch is able to perform its own operations independent of the other patches, or the processor it is assigned to. Having multiple patches per processor and message-driven scheduling helps improve performance in two ways: a) overlapping communication with computation and b) to achieve load balancing.

3.6 Load Balancing

Having more patches than the number of processors and ability to map them any processor help load balancing. Load balancing in NAMD is managed at two levels: a) initial mapping of patches to processors, *static load balancing*, and b) migrating patches as the load become imbalanced during the simulation, *dynamic load balancing*.

The `PatchDistrib` object invokes a partitioning algorithm to do the initial mapping of patches to processors. Currently, there are two options: a) strip partitioning which is a trivial algorithm that distributes patches to processors in a sequential order while trying to maintain the number of atoms per processor balanced, and b) a recursive bisection algorithm (`RecBisection` object, which divides the simulation domain into rectangular prisms rather than strips for better computation/communication ratio.

The dynamic load balancing is achieved by collecting timing statistics during the simulation, and use these statistics to adjust the mapping of patches. The `LoadBalance` object collects these statistics and recalculates the new mapping and then informs the `PatchDistrib`. The `PatchDistrib`

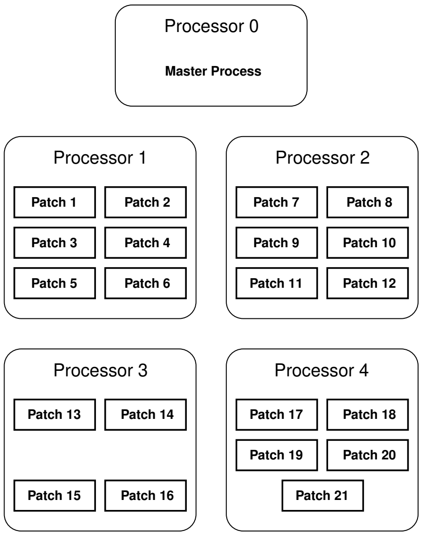

Figure 4: Example of patch assignment to processors

object then initiates the migration of patches. The dynamic load balancing is performed at every n cycle where n is defined by the user in the simulation configuration file.

In this version, although the code to collect the statistics and to do remapping is available, it is not tuned and optimized. Therefore, the dynamic load balancing feature will be fully available in future releases.

3.7 An overview of the control flow

In this section, a brief presentation of the flow of control from node level to the patch level will be presented. As it is described previously, NAMD is composed processes, which we will call it a `node`, and multiple patches per processes.

3.7.1 Node Level Control

NAMD spawns p processes at the beginning where each process invokes the node level algorithm (`Node` object). The skeleton code is shown in figure 5. The first thing to do is to read input files and initialize the system. The master node reads the input files and send the information to the other processes. After initialization, every process enters into time-step loop which invokes `doTimeStep` function of the `PatchList` object. The major objects that each node maintains are:

PatchList - controlling object for the set of patches reside on this processor.

Molecule - this object maintains the molecular structure of the system simulated (from `.psf` files).

SimParameters these are the global simulation parameters such as timestep size, cutoff values, etc.

PatchDistrib - this object performs spatial decomposition and maintains the mapping of patches to processors.

```
if (process id == 0)
     master_startup()
else
     client_startup()
for(timestep=begin to end)
     patchList->doTimestep(timestep)
}
```

Figure 5: Node level logic

```
doTimestep(timestep) {
     for(all patches) initialize patches
     while(all patches not done) {
          wait for a message
          extract destination patch
          patch->process_message(message)
     }
     if (end of cycle) redistribute atoms
}
```

Figure 6: PatchList logic

Output - this object produces the output as requested by user in various formats such restart files, DCD files etc.

3.7.2 PatchList

`Patchlist` object coordinates the patches that reside in this process. It handles message communication among patches transparently, i.e., patches send messages to other patches by using only the patch identifier without worrying in which process the destination patch resides. The `PatchList` object knows the mapping (by interacting with `PatchDistrib`), and forwards the message to the appropriate process. `PatchList` also schedules incoming messages and invokes the patches that the message is intended for. A skeleton code is shown in Figure 6.

3.7.3 Patch Structure

Each patch has the following objects or data structures:

atoms - the global indices of atoms in this patch (local atoms)

coordinates - the coordinates of local atoms

forces - the forces acting on local atoms

list of neighbors - the list of neighboring patch identifiers

set of force objects - these objects calculate forces for various types of interactions, such as bonded, electrostatics, constraint, etc. Each force object maintains necessary data structures for computing that type of interactions such as a pairlist for electrostatics or a bond list for bonded interactions.

integrator - this object computes the new positions after the force calculations done. It maintains the velocities of the local atoms.

3.7.4 Force Object Interface

Each force object has a set of interface routines that basically performs the following functions:

Local Initialization - Given the local atoms, set up any data structures that will be necessary for computing the local interactions of the force being calculated, and then calculate these forces. Examples of the data structures that may need to be build include a pairlist for electrostatics and a bond list for bonded interactions.

Neighbor Initialization - Given the local atoms and a set of atoms from another processor, set up any data structures necessary and then compute the forces. This is basically the same as `Local Initialization` except that the atoms involved in the interactions are now from another processor. Also for the bonded force object, this routine produces a list of local atoms that need to be sent to the neighboring patch so that it can complete its bonded calculations.

Local Force Calculation - Given the local atoms and that `Local Initialization` has been done, compute the forces due to local interactions.

Neighbor Force Calculation - Given the local atoms, a set of atoms from a neighboring processor and that `Neighbor Initialization` has been performed for this neighbor, comput the forces due to interactions with this neighbor.

Energy Calculation - Return the energy for this type of interaction for the current timestep. This is probably not as much of a calculation as it is a reporting of an energy sum that has been kept during all of the force calculations.

Clean Up - Clean up any data structures that were built. This may be accomplished by freeing and reallocating force objects at each reassignment.

3.8 Full Electrostatics

NAMD was designed to incorporate algorithms for calculating all electrostatic interactions for a molecular system such as the Fast Multipole Algorithm (FMA). There are two decisions that are critical to the interface of NAMD to these algorithms. The first is the complete separation of the particle decomposition of particles done by NAMD to compute the local forces from the decomposition used by the electrostatics algorithm. This leads to some inefficiency since a conversion from one decomposition to the other is necessary. However, it maintains a layer of abstraction between the two, so that NAMD is not constrained to one algorithm and vice versa. The second decision is to lessen the computational cost of computing full electrostatics by using a multiple timestep integration scheme. In this scheme, local electrostatics are directly calculated every timestep. Every k timesteps, the full electrostatic algorithm is used to compute electrostatic forces that are at a distance greater than the local interaction length. These forces are then added to the local forces during the next set of k steps. While these forces will become more and more inaccurate as k becomes larger, it is felt that the inaccuracies that are introduced are significately less than those introduced by the discretization to a finite timestep.

NAMD currently incorporates the Distributed Parallel Multipole Tree Algorithm (DPMTA) developed by the Scientific Computing group at Duke University. The following section describes the interface used to incorporate DPMTA into NAMD. This interface is representative for the incorporation of other algorithms for electrostatics or other calculations.

3.8.1 DPMTA Interface

The interface to DPMTA consists of two classes. `LongForce` is a force object that resides in each `Patch` and is responsible for forming the interface for NAMD. The other object is `FMAInterface` which has one representative in each `Node` object and is responsible for interfacing to DPMTA. A diagram of this arrangement is shown in figure 7.

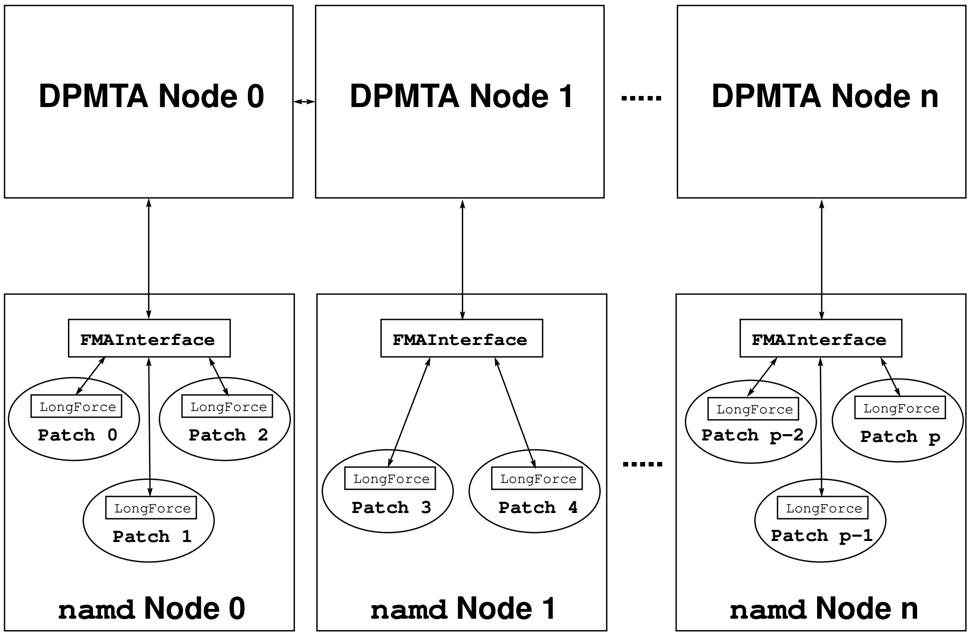

Figure 7: Schematic depiction of the NAMD to DPMTA interface

The way the interface works can best be understood by describing each cycle of computation. During the first timestep, each `LongForce` object gathers the coordinates from its `Patch` and deposits them in the `FMAInterface` object on its node. Once the `FMAInterface` object has all of the atom coordinates for its node, it forms arrays of coordinates and charges which are passed to DPMTA for the calculation of full electrostatics. The results of this calculation are returned to and stored in the `FMAInterface` object. These results are the forces acting on each atom due to the electrostatic interactions with ever other atom in the simulation. Next, these results must be returned to each individual `Patch` and the contributions from any electrostatic interactions within the local interaction distance must be removed, since these will be directly calculated during each timestep.

Each patch then proceeds to calculate the local forces as normal except that each set of coordinates received from neighboring patches are also passed to the `LongForce` object. Direct calculations of the electrostatic forces are performed and the results are stored. When all of the local interactions are calculated, the `LongForce` object then retrieves the forces due to all electrostatic interactions from the `FMAInterface` object. It then subtracts the local electrostatic forces that it has directly calculated to form the net long range electrostatic forces. It is these forces that are then contributed to the net force on each atom during each of next $k-1$ timesteps.

The current implementation of this interface is consists of a set of library routines that are compiled into NAMD and a separate executable that actually performs the DPMTA calculations. This separate binary (`dpmta_slave`) is started and communicated with via PVM. Thus, the current implementation is specific to the use of PVM as the underlying communication protocol. Also, the current implementation relies on the ability to start multiple processes on each node, which may not be the case on some MPP machines. As of this writing, DPMTA is restricted to an even power of 2 number of nodes. Since NAMD can run on any number of nodes, the number of `dpmta_slave` processes started is the largest power of 2 that is smaller than or equal to the number of nodes that NAMD is running on. Thus, if NAMD was run on 6 nodes, 4 `dpmta_slave` processes would be started.

The source code for DPMTA is available from the Scientific Computing group's WWW home page at `http://www.ee.duke.edu/Research/SciComp.html` or via anonymous ftp from the server `ftp.ee.duke.edu` in the file `/pub/SciComp/src/DPMTA_2.0.4.tar.Z`.

3.9 Design Decisions

This section contains design decisions that have been made for NAMD that don't fit into other sections. Right now, this section is just a list of these decisions that are kept here to make sure that they don't get lost. Hopefully eventually they will be part of more elegant design sections.

- NAMD will be written using C++ to facilitate the goal of having a modular program that is easy to modify and try new algorithms with. C++ provides a high degree of modularity and data encapsulation. A conscious effort has been made to reduce the use of any C++ features that could add overhead over the use of C, such as virtual functions.

- The output of messages to the screen, including warning messages and error messages, will be accomplished used I/O streams that are implemented in the class Inform. There are 4 global Inform objects per process that perform this task: `namdErr`, `namdWarn`, `namdInfo`, `namdDebug`. By using these objects rather than dircetly outputing messages, schemes such as the current one where all output is sent to the master process and then outputed can be implemented.

- The restart mechanism will consist periodically writing out the current positions and velocities in pdb format. Since NAMD will be able to start any simulation using pdb files to specify the initial positions and velocities, restarting a simulation will be no different that starting a new simulation.

- NAMD will take the same input files as X-PLOR. For a simulation, these inputs include: a pdb coordinate file, a psf structure file, one or more parameter files, and optionally a pdb velocity file.
- For the first implementation, all parameters will be stored on each processor. But the Parameter and Molecule objects will be written in such a way that this data could be distributed in the future.
- To determine initial velocities, NAMD will either take an initial temperature and assign random velocities to atoms to achieve this temperature, or read a pdb velocity files that specifies the initial velocity for all atoms.
- NAMD will create trajectory files in binary DCD format.
- NAMD will capable of feeding a network based connection to the program VMD. This involves periodically gathering all of the coordinates and sending them via a TCP/IP connection to VMD.
- NAMD should be able to operate using either the communicate class or via the Charm++ coordination language. The parallel control structure of NAMD should be written in a way that facilitates this.
- NAMD obtains all of its simulation parameters (i.e. input file names, timestep length, etc.) from a configuration file that is specified on the command line. If a '-' is passed rather than a file name, the configuration file is read from stdin.
- There will be typedefs for `Real` and `BigReal`. These will be used in place of float or double to allow flexible switching of the accuracy. By default, `Real` is typedef'd to float and `BigReal` is typedef'd to double.
- All tags for messages will be `#define`'d in the source file `common.h`.
- The `Communicate` object will implement a kind of prioritization of messages in that messages that are from other processors will be retrieved before messages that are sent from the same processor. This will insure that in situations where an incoming message will require a message to be sent in response, cases that require interprocessor communication will be handled before those requiring only local communication.
- Every message involved with transferring data during a given timestep will be of the form:
  1. Destination patch id
  2. Timestep identifier indicating the timestep that this data is for
  3. Sending patch id
  4. Data specific to this message
- The routine `send_msgs()` should be called at the end of mid-cycle finishing timesteps rather than at the begining of new timesteps so that the communication time of these messages can be overlapped with the computation being done by other patches on the processor.
- Currently, all patches on a node are required to be working on the same timestep. Nodes may continue to the next timestep without synchronizing with other nodes, but patches may not. This decision may be limiting performance and should be revisited later.

3.10 MDComm

3.10.1 Overview:

MDComm is the communications component of MDScope. It is built on top of the RAPP software package (written at NCSA), which is a set of library routines and associated programs that simplify the process of building and maintaining interactive client/server applications. In MDScope, MDComm transfers the simulation information from the molecular dynamics program NAMD, to the graphics program VMD so the information can be visualized as it is computed. In the other direction, MDComm is used to transfer user commands from VMD to NAMD in order to modify the simulation interactively. At present this is used to apply user defined forces to specific atoms in the simulation, but in the future many more types of commands will be available.

MDComm uses TCP/IP (BSD sockets) for the communications which allows the simulation and visualization programs to execute on different machines. Thus, the simulation can be run on a remote supercomputer or workstation cluster while the visualization occurs on a graphics platform. Additionally, using MDComm, VMD can maintain multiple simultaneous connections to different simulations.

The MDComm connection does not have to be maintained during the full simulation. Instead, the NAMD process can be detached from VMD and left to run on its own. The connection can later be reestablished, and the current state of the simulation viewed and modified. This is useful when running jobs which take days, weeks, or even months to finish as it provides a simple way to check on the status of the simulation.

For more information about MDComm, see the MDComm home page at:
`http://www.ks.uiuc.edu/Research/mdcomm/`

3.10.2 How MDComm organizes the different parts of MDScope:

One program, rappd, must be running on the simulation machine. This maintains a list of programs which can be started on the given machine as well as a list of all programs which are currently running. It does not start any programs or transfer data between VMD or NAMD. Rather, it provides the information needed to start or connect to a simulation.

On the visualization side, VMD contacts `rappd` and receives a list of available programs along with the options they need for startup. The VMD user enters the information into VMD and tells the program to start a new NAMD task.

VMD rexec's a `namdd` (note the double d), which is the daemon process for an instance of namd. The namdd does four things:

- it collects the configuration information for NAMD into a file then starts it;
- it maintains a list of static information, such as the number of atoms, bonds, and residue names. This is done so VMD can easily reattach to the simulation without having to request the information from namd again
- it notifies `rappd` when NAMD has started and finished
- it passes user commands (such as the interactive forces) to NAMD Since the communications between the visualization and simulation machines can have different numeric representations, `namdd` also performs XDR decoding at this step.

After NAMD starts and successfully reads the input files, it contacts the namdd and transfers over the static information.

At each timestep, it checks to see if the timestep data (coordinates, energies, and patch information) should be transferred to VMD. (Actually, it sends the data to the `namd_consumer` on the visualization machine, which does the XDR decoding and buffers the resultant to VMD.)

Also at each timestep, it checks if a user command has come from VMD. Currently, only one commands is allowed, which sets a user-defined force vector for a set of atoms.

3.10.3 Implementation:

There are only two functions needed for the basic interface. They are:

```
mdcomm_send_static(rapp_active_socket_t *, void *)
```

This sends the static information to the namdd. The message is packed in a specific way which can be unpacked by `mdcomm_recv_static` in VMD.

```
mdcomm_send_dynamic(rapp_active_socket_t *, void *, void *)
```

This sends the dynamic information to the `namd_consumer`. Again, the message is structured in such a way that VMD can understand it with the `mdcomm_recv_dynamic` function.

The distributed version of NAMD does not yet support the interactive user commands. (The interactive code diverged slightly from the main NAMD code. The patches needed to make the standard distribution work with the steering forces will be distributed when the two code sources have been merged.) The following describes how the internal version currently works.

When MDComm receives a force message over the rapp connection (a user message with the tag `MDCOMM_FORCE_CMD`) it calls the function `mdcomm_app_force_cmd` on the start-up node of the NAMD job (node 0). This calls the command `mdcomm_transfer_vmdForceData`, which sends a standard NAMD message (of type `VMDFORCETAG`) to all the other nodes. The data is stored on a per atom basis in the `vmdForceData` array. When the `PREINTEGRATE` message is received, the value of `vmdForceData` is added to the sum of the forces applied to an atom.

4 Working with NAMD

4.1 Working Environment

(**NOTE:** Much of the information in this section is specific to our development setup here at UIUC. Outside readers should ignore the specifics of much of this section, but can still use the information to see how the files were created, how to set up a similar environment elsewhere, etc.)

NAMD files are kept in a main RCS directory, from which you check out files that you will change, and later check back in when complete. You must first set up a working directory structure, with links to the main RCS directory.

Setting up your working directory

On the HP's a group '`namd`' has been created (GID=10060) for this project. An RCS-based working directory setup has been created in the directory `/usr/local/src/namd`, with group `namd` write permissions. To create your own working directory, do the following:

1. `cd /usr/local/src/namd`
2. `setup_user <your username>`

This will create a working directory for you, including a set of subdirectories for src and obj files. A copy of the makefile and source files will be checked out.

Compiling NAMD

The source files are in the directory `<working dir>/src`, along with the makefiles. These makefiles are set up to handle different types of architectures, for example HPUX9 and IRIX5. The variable ARCH at the beginning of the file `Makefile` indicates for which architecture you are compiling. To compile, just run 'make'. These makefiles are described more completely in section 4.1.

Adding new files

All source files in the RCS should have an RCS header; templates for the various type of files (.c or .C, .h, .tex, and makefiles or shell scripts) are in the directory `/usr/local/src/namd`. Place a copy of these templates at the beginning of the file; keywords found within $ pairs will then be replaced by the RCS program with their values when the file is checked out. The 'Description' section in the template should also be filled with a text description of what the file contains.

To add new files into the RCS directory, add the proper header and execute the following commands:

1. `rcs -i -c"`*comment leader*`" -L -a`*user1,user2*`,... <file1> ... <fileN>`
2. `ci -u <file1> ... <fileN>`

where "*comment leader*" is one of the following, based on the type of file:

- `" * "` for .h, .c, and .C files.
- `"% "` for .tex files.

- `"# "` for makefile files, and shell script files.

When adding a new file, you will also have to add it to the file listing in `Makedata.files`, and possibly update `Makedata.depend` (see section 4.1).

WARNING: make sure you check in/check out files on the HP's ONLY ... do NOT use the SGI's or NeXT's to do this. The HP's RCS program formats files differently (for some reason), and so mixing the two programs will result in an incorrectly-formatted RCS file. Once a files is checked out, you can edit it on any platform you wish, but only use the HP's RCS commands.

Makefile structure

Setup The main Makefile is in the 'src' directory; it includes several auxiliary files:

- Makedata.HPUX9 and Makedata.IRIX5 contain architecture-specific definitions for HPUX or SGI. The architecture to use is defined at the top of Makefile.
- Makedata.files is where all the source files to compile are listed, along with other files we may need later.
- Makedata.depend lists the file dependencies ... on the SGI's there is a nice way to get the compiler to do this for us, but sadly the HP's do not have this capability. Maybe we can do it once on the SGI's and use the resulting file on the HP's, I'm not sure yet.

That's it, I hope this is somewhat less complicated than Andreas's setup. The obj files all get copied to the directory src/objfiles/ARCH, to facilitate compiling on multiple architectures.

Other commands Other useful makefile commands which are available:

- 'clean' - yeah, pretty obvious.
- 'doc' - latex the documentation, including this document.
- 'co' - checks out the most recent version of ALL the files; if any are locked by you, a message will be printed.
- 'co.h', 'co.src', 'co.make', 'co.doc' - checks out most recent version of just the specific files in the category indicated.

4.2 Source Code Style Conventions

4.2.1 File names

Source files

- All C++ files should start with capital letters, have all words capitalized, and not use hypens or underscores. They should end with '.C'. Example: Communicate.C
- All C files should have only lower case letters in their name, and end with '.c'. Example: util.c

Header files All .C and .c files should have a corresponding .h file. With C++, though, it will of course happen that there will be .h files with no .C file. Put all code in a header file between a

```
#ifndef HEADER_IDENTIFIER
#define HEADER_IDENTIFIER
...
#endif
```

construct.

Make files The main Makefile includes several auxiliary files, as described in 4.1. All data files for the main makefiles should be named as `Makedata.<name>`, i.e. `Makedata.depend`.

Documentation files Documentation files, including this guide, are in the 'doc' directory. Latex files should have an RCS header prepended.

Make sure there is a link in the 'doc' subdirectory of your working dir to the main figures directory (of which there is only one, we do not put figures in the RCS). If there is not, do the following:

ln -s /usr/local/src/namd/figures figures

Figures (such as eps files) should be place in the main figures directory. Refer to figures in the documentation files as `figures/<filename>`.

4.2.2 Source code names

Class names C++ classes should all go in separate files, if possible, and have the same name as the file they are in. Thus, use the same naming convention for classes as for files.

Class member names Variables which are members of a C++ class should start with a lower case letter, have every OTHER word in the name capitalized, and not use hyphens or underscores. An exception is single-word variables, which may start with a capital if preferred. Examples: bondLength, coorFileName, Size.

Global variables Variables global to NAMD (error and info message objects, etc.) should have 'namd' prepended to them. Example: namdInfo.

Class function names Functions which are members of a C++ class should all be in lower case, and use hyphens to separate the words in the name. Example: real bond_length(Bond b)

Accessor functions which get and set a single item in a class should both be named the same, one with the relevant return type and no arg, one with relevant arg and either void return type, or success (i.e T/F) return type. Example: float length(void); and void length(float);

Accessor functions which get/set multiple items should have one named 'set_item' and the other 'get_item', where 'item' is whatever is being accessed.

Global function names Functions global to NAMD should have 'NAMD_' prepended to them. Example: NAMD_die();

Macros Macros should be in ALL CAPS ... 'nuff said.

4.3 Configuration Options

NAMD reads a configuration file to figure out what to do. This file specifies where the input structure is located, the file containing the force parameters, the initial velocities, and so on. The format of the configuration file is `keyword = value`. The "=" sign is optional. The keywords are case insensitive. Comments are allowed. They start with the "#" symbol and end at the end of the line. If there is a line in the file which is not understood by NAMD it will give a warning, print that line, and continue. Here is an sample configuration file for a simulation of bacteriorhodopsin.

```
coordinates = bR.pdb   # the bR568 structure
structure = bR.psf
  # read in the different energy parameter files
parameters = parmalh3x.pro
parameters = param19.sol
parameters   parambr_H_n.est
temperature 300        # assume a Maxwellian velocity distribution at 300 K

timestep = .75         # take small ( 3/4 femtosecond) time steps
numsteps = 1000
stepspercycle = 20     # specify the number of steps between reassignment
DCDfile = bR.dcd       # write to the DCD file after every 100 steps
DCDfreq = 100

  # adjust the parameters for the electrostatic cutoff calculations
cutoff = 8.5

  # prefix name for the coordinate and velocity output files
outputname = bRsimulation

  # prefix name for restart coordinate and velocity files
restartname = bRrestart
```

Some of the configuration variables have default values. For instance, the default step size is 1 femtosecond. Some of the variables can only be specified once. Others, such as the parameter file, can be listed several times. Some of the options are mutually exclusive. If there is a conflict or error, NAMD will print an error message and stop. See the NAMD *User Guide* for the list of options.

5 Implementation

This section provides a detailed description of all aspects of the NAMD implementation. This includes everything from the high level program structure to detailed description of each class that is used within the program.

5.1 Program structure

Initialization

The first object created in the Communicate object, which makes sure the program is running on all other nodes in the parallel machine. When constructed, this object knows how many nodes are available, and a node number for each node. Node 0 is the *master* node, with nodes numbered 0 ... N-1 if there are N total nodes.

The master node reads in all data, verifies structure and parameters, parses configuration file, and sends data to compute nodes. All nodes keep a full copy of the parameters and the complete structure of molecule.

More to come ...

5.2 Global Definitions

Global items are defined in `common.h`; global variables and functions are there declared `extern`. The global functions are in `common.C`; global variables are actually declared in `namd.C`.

Global functions

- `NAMD_title(void)`: Print out a title message. Really should only be called by the master node.
- `NAMD_check_messages(void)`: This checks the Inform message objects if they have any new messages, which are displayed if available. This should be called by the master node *only*, and should be called periodically, in some form of event loop.
- `NAMD_quit(void)`: Exits normally from NAMD, with a 0 exit code.
- `NAMD_die(char *)`: Exits abnormally, with the given error message.
- `BigReal NAMD_random()`: Returns a random number between 0 and 1.

Global variables

- `Inform namdInfo`: Inform object which takes informative messages and forwards them to the host node, where they are displayed. See section 5.3.14 for how to use the Inform object.
- `Inform namdWarn`: Inform object for warning messages. Warning messages mention unusual occurances, things that do not warrant stopping the simulation, but which might be useful to the user.
- `Inform namdErr`: Inform object for error messages. Error messages are typically items that prevent a simulation from starting or continuing.

- `Inform namdDebug`: Debugging messages. When debugging is enabled, messages sent to this object will be printed. Use this to display general debug information, not stderr or cerr.
- `Communicate *comm`: Pointer to the main communication object, used to send messages between nodes.
- `Node *namdMyNode`: Node object present on each processor.
- `int namdNumNodes`: Number of nodes involved in the simulation.

Global defines

- `PI, TWOPI, ONE, ZERO`: Defined to set the proper accuracy; defined up to 15 places for 'double' accuracy.
- `TRUE, FALSE, YES, NO`: Pretty obvious.
- `BOLTZMAN`: Boltzman's constant

Global typedefs

- `Real` - used for all computation storage, where a 'float' or 'double' might be used. Allows us to set the desired accuracy/storage size preference.
- `Bool` - for true/false items.

5.3 Class Descriptions

This section contains descriptions for the different major classes used in NAMD. These descriptions include the class name, interface, and examples of use.

5.3.1 AngleForce

Purpose:

The `AngleForce` class is used to calculate the forces and energies due to angular bonds in a molecule. There is a `AngleForce` object present in every patch and it is responsible for calculating the angle interactions between local atoms as well as between local atoms and atoms from neighboring patches. This object is very similar to the `BondForce` object with some additions to deal with the fact that angles can be split between three patches, unlike bonds that can only involve two patches.

Files:
`AngleForce.h,AngleForce.C,structures.h`

Constructor:
`AngleForce(Patch *parentPatch, PatchList *parentList)`
The parameters `parentPatch` and `parentList` allow the object to know who owns it.

Destructor:
`~AngleForce()`

Method of Use:

There are a small set of public routines to access this object. `initialize_timestep()` is the first of these and it simply gets the object ready for a new timestep. The function `local_init()` is used to initialize the object for calculating the local angle interactions. It is used during the first timestep of each cycle. The function `local_force()` is then used for all the other timesteps in a cycle to calculate the forces and energies due to local angles. The function `neighbor_init()` is used to initialize the object for interactions between this patch and a neighboring patch. It builds all the data structures necessary for calculating interactions with this neighbor, and then computes the interactions. If the neighbor being initialized is one that sends all of its coordinates, then this function also returns the local indexes of atoms that need to be sent to this neighbor as bonded coordinates. The function `neighbor_force()` is used to calculate the forces between this patch and neighbor patch during mid-cycle timesteps. The function `get_energy()` returns the accumulated energy for the current timestep. While the forces due to a angle that is shared between patches are calculated by both patches, only the patch that owns the first atom in the angle will calculate the energy associated with this angle. This prevents angle energies from being counted more than once.

Internally, there are a few more details to what happens. The first time one of the initialization functions is called, the function `build_angle_list()` is used to build two lists of angles. The first is a list of local angles, that is, angles where all three atoms are local to this patch. This list remains unchanged and is used to calculate local forces and energies. The second list is a list of all of the angles that involve at least one local atom and at least one atom from another patch. This is the unassigned list. As neighbors are initialized, a list of angles between the local patch and this neighbor patch is formed. This list of angles is constructed by scanning the unassigned list and linking angles that belong to this neighbor into the list for this neighbor. But there is a complicated case than can arise when the three atoms involved in an angle are owned by three different patches. If this is the case, then the angle must appear in multiple neighbors lists, and space for buffering coordinates must be provided to hold the coordinates from the neighbor whose message arrives first. This is accomplished by storing all of the data for a particular angle in a structure that is then pointed to by the nodes of each neighbors linked list. This information structure may be pointed to by multiple neighbors. An illustration of this is shown in Figure 8. Once this list is built for a neighbor, the forces and energies for interactions between this patch and the neighboring patch can be determined by just walking down the list. Also, once all the neighbors have been initialized, the unassigned list should always be empty.

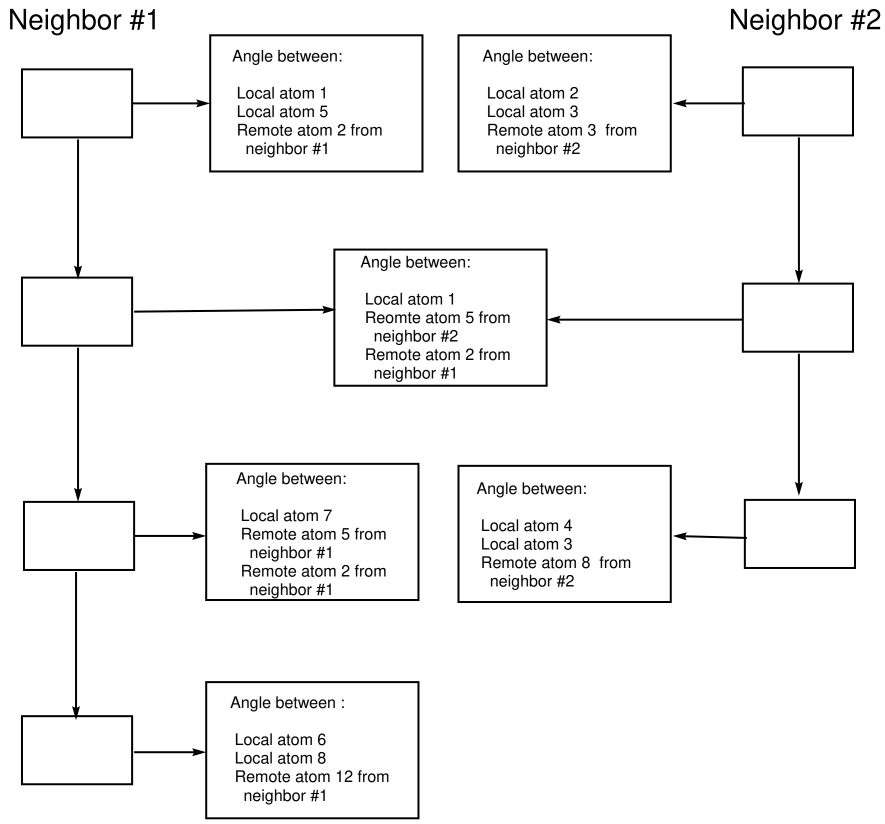

Figure 8: Example of angle lists with angles that are shared between 1 and 2 neighboring patches

For the first implementation, there are certain aspects of this object that are questionable. Currently, the most questionable is the way that the angle list is constructed. It currently requires a list of the angles that each atom is involved in to be stored. This is fairly expensive in terms of memory, but quick. This scheme should be revisited some time later.

Public functions:

- `void initialize_timestep()`
  Get the object ready for a new timestep.
- `void set_recycle()`
  Tell the object that the atoms owned by this patch haven't changed, so the angle list from the previous cycle can be reused.
- `BigReal get_energy()`
  Get the angle energy that has been accumulated during this timestep.
- `void local_init(int numAtoms, int *glbInds, Vector *x, Vector *f)`
  Given the local positions and force vectors for this patch, first initialize the object to perform local interactions and then calculate these interactions adding the forces to the force vector given. This function should only be used during the first timestep of a cycle.
- `void local_force(Vector *x, Vector *f)`
  Calculate the local angle forces for this patch given the position and force vectors. This function should be used during every timestep of a cycle except the first timestep. If this function is called before `local_init()` then bad things will happen.
- `void neighbor_init(int nid, int nsize, int nlocal, int nAtoms, int *glbInds, Vector *localx, Vector *remotex, Vector *f, IntTree *sendtree=NULL)`
  This function initializes the data structures for calculating angle interactions between this patch and the neighboring patch indicated by `nid`. Once the data structures are built, then the forces and energies for these interactions are calculated and the local forces are updated. `nAtoms` is the number of coordinates received from this neighbor, `glbInds` are the global atom indexes of the coordinates received from the neighbor, `localx` are the local position vectors, `remotex` are the position vectors from the neighbor, `f` are the local force vectors to be updated, and `send_tree` is an `IntTree` object that is used to store the atom indexes of atom coordinates that need to be returned to this neighbor as bonded coordinates. If the neighbor only sends bonded coordinates, this will be `NULL`. This function should only be called once per neighbor and only during the first timestep of each cycle.
- `void neighbor_force(int nid, Vector *localx, Vector *remotex, Vector *f)`
  Calculate the angle interactions between the local patch the the neighbor indicated by `nid`. It does not matter whether the patch `nid` is passing us all coordinates, or only local coordinates. It does matter that initialization for this neighbor has already taken place using the routines above.

Private functions:

- `void recycle_lists()`
  This function is used when the atoms in a patch don't change from one cycle to the next. In this case, rather than having to remake the lists of angles, we will recycle them. The list of local angles can be reused as is. The neighbor lists of angles still need some work though, since we don't know whether our neighbor lists have remained the same. So basically instead of building the unassigned list from scratch, we will rebuild it from the old neighbor lists. This should still be much more efficient.

- `void build_angle_list(int numAtoms, int *glbInds)`
  This function uses the structural information from the local `Molecule` object to build two lists of angles. The first is a local angle list that contains angles that contain only local atoms. The second is the `unassignedBonds` list. This is a list of all the angles that involve at least one local atom and at least one remote atom. This list is then searched as neighboring coordinates come in. All the bonds on this list should be assigned by the time that we have receieved coordinate from all our neighbors.

5.3.2 BondForce

Purpose:
The `BondForce` class is used to calculate the forces and energies due to linear bonds in a molecule. There is a `BondForce` object present in every patch and it is responsible for calculating the bonded forces between local atoms as well as between local atoms and atoms from neighboring patches.

IMPORTANT NOTE: Since the bonded interactions only require interactions between two neighboring patches, the `BondForce` object is the only bonded force object that does not do duplicate calculations on patches. It instead returns the forces to the neighbor along with the electrostatic forces.

Files:
`BondForce.h,BondForce.C,structures.h`

Constructor:
`BondForce(Patch *parentPatch, PatchList *parentList)`
The parameters `parentPatch` and `parentList` allow the object to know who owns it.

Destructor:
`~BondForce()`

Method of Use:
There are a small set of public routines to access this object. `initialize_timestep()` is the first of these and it simply gets the object ready for a new timestep. The function `local_init()` is used to initialize the object for calculating the local bonded interactions. It is used during the first timestep of each cycle. The function `local_force()` is then used for all the other timesteps in a cycle to calculate the forces and energies due to local bonds. The function `neighbor_init()` is used to initialize the object for interactions between this patch and a neighbor patch and is used only during the first timestep of a cycle. The function `neighbor_force()` is used to calculate the forces between this patch and a neighbor patch during mid-cycle timesteps. The function `get_energy()` returns the accumulated energy for the current timestep.

Internally, there are a few more details to what happens. The first time one of the initialization functions is called, the function `build_bond_list()` is used to build two lists of bonds. The first is a list of local bonds, that is, bonds where both atoms are local to this patch. This list remains unchanged and is used to calculate local forces and energies. The second list is a list of all of the bonds that involve one local atom and one atom from another patch. This is the unassigned list. As neighbors are initialized, a list of bonds between the local patch and this neighbor patch is formed. This list is constructed by scanning the unassigned list and taking bonds from this list and placing them into the specific neighbors list. Once this list is built for a neighbor, the forces and energies for interactions between this patch and the neighboring patch can be determined by just walking down the list.

There were several decisions made during the design of this object that should be reconsidered later. The main one is the way in which the bond list is built. Currently this relies on having a list of bonds that each atom is involved in. This is quite fast, but expensive and non-scalable in terms of memory usage. If the structure of the molecule were distributed rather than stored on every node, this would become more reasonable.

Functions:

- `void initialize_timestep()`
  Get the object ready for a new timestep.
- `void set_recycle(void)`
  No atom reassignments were made involving this patch, so tell the object to reuse the bond list.
- `BigReal get_energy()`
  Get the bonded energy that has been accumulated during this timestep.
- `void local_init(Vector *x, Vector *f)`
  Given the local positions and force vectors for this patch, first initialize the object to perform local interactions and then calculate these interactions adding the forces to the force vector given. This function should only be used during the first timestep of a cycle.
- `void local_force(Vector *x, Vector *f)`
  Calculate the local bonded forces for this patch given the position and force vectors. This function should be used during every timestep of a cycle except the first timestep. If this function is called before `local_init()` then bad things will happen.
- `void neighbor_init(int nid, int nAtoms, int *glbInds, Vector *localx, Vector *remotex, Vector *localf, Vector *remotef);`
  This function initializes the data structures for calculating bonded interactions between this patch and the neighboring patch indicated by `nid`. Once the data structures are built, then the forces and energies for these interactions are calculated and the local forces are updated. `nAtoms` is the number of coordinates received from this neighbor, `glbInds` are the global atom indexes of the coordinates received from the neighbor, `localx` are the local position vectors, `remotex` are the position vectors from the neighbor, `localf` are the local force vectors to be updated, and `moretef` are the force vectors for the neighboring atoms to be updated. This function should only be called once per neighbor and only during the first timestep of each cycle.
- `void neighbor_force(int nid, Vector *localx, Vector *remotex,Vector *localf, Vector *remotef)`
  Calculate the bonded interactions between the local patch the the neighbor indicated by `nid`. It does not matter whether the patch `nid` is passing us all coordinates, or only local coordinates. It does matter that initialization for this neighbor has already taken place using the routines above.

5.3.3 Collect

Purpose:
The Collect class gathers the useful data produced by the patches. This data includes sum of various energy terms over all atoms, positions and velocities of each atom. The current implementation accumulates the data on the master node, then invokes the output object to report the data. The collection of energies is performed at every time step. Positions and velocities are collected at some frequency determined by the user. A spanning tree is employed for the sum of energies to yield better scalability. Currently, positions and velocities, however, are sent to the master node directly.

Derived From:
Nothing.

Files:
`Collect.h, Collect.C`

Constructor:
`Collect(int node)`: Initializes the spanning tree, and creates the output object only on the master node.

Destructor:
`~Collect(void)`

Method of Use:
An instance of the Collect object exist on each node. The constructor builds the spanning tree which is used to propagate the energy data. The interaction between the collect object and the rest of the namd program is as follows: At the beginning of each timestep, the Node object invokes the collect object on the same processor for time-step initialization. During the time-step, each patch then deposits their data, when it is ready, to the collect object. At the end of the time-step, the PatchList object tells the collect object to propagate the data collected so far (at this point all the patches contributed their data). The collect object then carries out the gathering of data. For energy collection, the collect object combines the partial results from its children (in the spanning tree) and propagates the combined result to its parent. Collect object sends the positions and velocities directly to the master node. Once the data assembled on the master node, then the output object is invoked

Defined Constants:
`maxEnergyCount` is the number of energy terms. Currently, there are nine of them: bonded, angle, dihedral, improper, electro-force, vdw, constraint, kinetic, and efield.

Private functions:

- `send_energy(void)`
  This function is a private function. It performs the spanning tree based propagation of the energies. On the master node, the output object is invoked.

- `send_coor_vel(Message *outmsg, int tag)`
  This private function carries out propagation of positions and velocities. On the master node, the output object is invoked.

Public functions:

- `void init_timestep(int timestep)`
  Initialize the collect object for a particular a time-step. This includes initialization of data area for partial results, and determining if the time-step is ok for collection of positions and velocities.

- `void energy(BigReal *energy)`
  Patches deposits their energy data through this function by supplying an array of energies. Each energy value is the sum over all the atoms that the patch owns.

- `void coordinate(int timestep, int n, int *globalIndex, Vector *coor)`
  Patches deposit their positions together with the list of global index of atoms that they hold by invoking this function.

- `void velocity(int timestep, int n, int *globalIndex, Vector *velocity)`
  Patches deposit their velocities together with the list of global index of atoms that they hold by invoking this function.

- `void propagate(int timestep)`
  This function triggers the process of collection across processors.

Derived Classes:
None.

5.3.4 Communicate

Purpose:
Abstract base class used to send messages between nodes of a parallel machine. Also responsible for initializing communications, and verifying that processes are running on all available nodes. Users can choose to have all messages sent immediately, or defer sending until a later time.

Note that this object is currently under construction (heh, heh).

Derived From:
Nothing.

Files:
`Communicate.C, Communicate.h`

Constructor:
`Communicate(void)`

Destructor:
`~Communicate(void)`

Enumerations:

- `CommError : NOERROR, ERROR, NONODES, NOSEND, NORECEIVE`
- `CommDir : SEND, RECEIVE`
- `SendMethod : NOW, WAIT`

Method of Use:
To send data to another node, first a Message object is created, which is loaded with the data to be sent (see section 5.3.21). Then the `send` routine is called with a pointer to the Message. This Message object is not copied, so the user must `new` a Message object before sending it via `send`. If a message is sent sucessfully, the Communicate object will automatically free up it's storage space; if it cannot be sent, the user is responsible for freeing the space.

Each message sent to another node must have a user-supplied tag to uniquely identify the message for the receiver. This tag should be greater than or equal to 0. Some specific tags are used by other components of NAMD, such as the Inform messages (tags 1000, 2000, 3000, and 4000). When receiving a message, you can ask to receive a message from a given node with a given tag, or specify a *wildcard* for either node or tag, by asking for a message with node = -1 or tag = -1.

The Communicate object can send all messages as soon as they are provided, or wait until the routine `send_all()` is called, which will send out all cached Message objects. This is quite often preferrable when several messages are to be sent to the same node; the Communicate class can combine these into a single message which is then broken into individual Message's by the receiver. By default, the send method is `NOW`, which means when a message is requested to be sent, it is indeed sent over the network. If the send method is `WAIT`, the message will be stored but not actually sent until `send_all()` is called. The send method can be set via the `send_method` routine. Also, there is a `send_now` function which will send a message NOW, regardless of the current send method setting.

To receive a message, the `receive` routine is called, which returns a new instantiation of a Message object if a message has arrived, or `NULL` if no messages are available. Once a message has been successfully `received`, the user must retreive data from it, and `delete` it.

A message can also be broadcast to all other nodes, or to all nodes including the sender. Broadcasts always occur immediately; they are not combined with other messages.

Functions:

- `void print_comm_stats(void)`
  Print out monitoring information about message traffic.
- `int nodes(void)`
  Returns total number of nodes available.
- `int this_node(void)`
  Return which node the process is on; nodes are numbered 0 ... nodes() - 1.
- `CommError errorno(void)`
  Returns the current error status, which is set after a send or receive call.
- `int debug(void) || void debug(int)`
  Query or set the debug flag; when debugging is set, several debugging messages are printed to the console during communication.
- `void send_method(SendMethod sm)`
  Set the current sending method, either `NOW` or `WAIT`.
- `SendMethod send_method(void)`
  Query what the current sending method is, `NOW` or `WAIT`.
- `int send(Message *msg, int node, int tag, int delmsg = TRUE)`
  Send the given Message object. Whether it is actually sent at the time the routine is called depends on the current SendMethod. If delmsg is not given or true, the Message is freed after sending.
- `int int send_all(void)`
  Sends out all yet-unsent messages, grouping together into single messages all individual messages destined for the same node.
- `int send_now(Message *msg, int node, int tag, int delmsg=TRUE)`
  Send the given Message object NOW, regardless of the current send method setting. If delmsg is not given or true, the Message is freed after sending.
- `Message *receive(int& node, int& tag)`
  Receives a message from the given node with the given tag. Node and tag may each be (-1), in which case they are wildcards and match anything. Returns a new Message object, for which the user is responsible for `delete`ing.

- `int int broadcast_all(Message *msg, int tag)`
  Sends (immediately) the given message to all nodes *including* the senders node, with the specified tag. Returns the number of nodes actually sent to.

- `int int broadcast_others(Message *msg, int tag, int delmsg=TRUE)`
  Sends (immediately) the given message to all nodes *except* the senders node, with the specified tag. Returns the number of nodes actually sent to. If delmsg is not given or true, the Message is freed after sending.

- `int add_node(void *id)`
  Adds a new node to the parallel machine. Since several different ways of specifying the id of the node are possible, the id is given as a pointer to void. Returns the new node number, or -1 if there is an error.

- `int get_tids(void)`
  Get the PVM tids for FMA. This really shouldn't be a virtual function of this class, but it has to be in the current setup.

Derived Classes:

- `CommunicatePVM (CommunicatePVM.C, CommunicatePVM.h)` - PVM 3 version of Communicate. Will determine state of virtual machine, spawn processes on other nodes, and verify all nodes are communicating. If PVM is not running, or there are problems, simply assumes there is just one node.

5.3.5 ConfigList

Purpose:
Reads information from either a configuration file or standard input and creates a searchable data base. Entries in the file are of the form "keyword" = "data". The equals sign is optional. Blank lines are ignored, as are comments, which start with a "#". Multiple entries with the same keyword are allowed. Keywords are not case dependent.

An alternate form of entry is

```
keyword = {
line 1
line 2
...
line n
}
```

Each line is added to the keyword data base, including spaces. This method is mostly useful for multiline inputs. Again, the equals sign is optional.
Derived From:
Nothing.

Files:
`ConfigList.h, ConfigList.C`

Constructor:
`ConfigList(char *filename)` where `filename` is the configuration file name. This opens the file, reads and parses the information within, and closes the file. The "filename" for standard input is a hyphen "-" (that's a somewhat standard Unix idiom). If it could not open the file, then the member function `okay( void)` is set to `FALSE`, otherwise it is `TRUE`. Lines that could not be parsed are written to namdWarn.

Destructor:
`~ConfigList(void)`

Method of Use:
The constructor must be called with a filename:

```
ConfigList config("bR.namd");
if (!config.okay())
    NAMD_die("Could not read configuration file.");
```

Information about a keyword can be retrieved with the "find" function. This returns a linked list of the type StringList. For example, to get a list of all the parameter files (which contain the energy constants).

```
StringList *tmp, *params = config.find("parameters");
if (!params)
```

```
      NAMD_die("No paramater files listed.");
   for(tmp = params; tmp != NULL; tmp = tmp -> next )
         cout << tmp ->data << '/n';
```

Typedefs:
`StringList`: a structure containing two data elements: `data` is a `char *` to the string value being stored and `next` is a `StringList *` to the next element of the list. The list is NULL terminated. This typedef has its own constructor and destructor. The constructor takes a `char *`, creates space for the string, and copies it. The destructor deallocates that space. This structure is used to return the value(s) for a given parameter name.

Functions:

- `Bool okay( void)`
  Returns TRUE if the file could be opened, FALSE otherwise.
- `StringList *find(char *keyword)`
  Returns a linked list of StringList containing all the data associated with the `keyword`. If there are no entries in the configuration file that match `keyword` then the returned value is NULL. The order of the elements is that same as the order in the file. This function does not allocate any new space, so do not attempt to delete it.
- `ConfigList::ConfigListNode *head( void)`
  Returns the pointer to the internal data structure. Nothing now uses this function. A better interface should be written, but there is not now any reason to do so.

5.3.6 ConstraintForce

Purpose:
The `ConstraintForce` class is used to calculate the forces and energies due to harmonic atom constraints as described in section 2.1.8.

Files:
`ConstraintForce.h,ConstraintForce.C`

Constructor:
`ConstraintForce()`

Destructor:
`~ConstraintForce()`

Method of Use:
Because of there are no interactions with other patches, this class has a simple subset of the generic force object interface. The function `initialize_timestep()` is used to begin a timestep. The function `init()` is used to initialize and compute forces during the first timestep of a cycle and the function `force()` is used to compute energies and forces during mid-cycle timesteps. The function `get_energy()` is used at the end of a timestep to obtain the energy computed during the current timestep.

Implementation:
In the function `init()`, a list of all the atoms on the current patch that are constrained is made along with the appropriate constants and positions. These values are then used for the remainder of the cycle to compute the constraint forces.

Functions:

- `void initialize_timestep()`
  Get the object ready for a new timestep.
- `BigReal get_energy()`
  Get the constraint energy that has been accumulated during this timestep.
- `void init(int numAtoms, int *atomInd, Vector *x, Vector *f)`
  Initialize the force object for the next cycle and compute the energies and forces for the first timestep. `numAtoms` is the number of atoms on the current patch, `atomInd` is a list of the global indexes for these atoms, `x` is an array of position vectors for these atoms, and `f` is an array of force vectors for these atoms. The forces do to the constraints will be added to the array `f`. If necessary, compute the current reference position for the atom specified for "moving constraints" option.

- `void force(Vector *x, Vector *f)`
  Calculate energies and forces for a mid-cycle timestep. `x` is an array of position vectors for the atoms on this patch and `f` is an array of force vectors for these atoms. The forces do to the constraints will be added to the array `f`.

5.3.7 DihedralForce

Purpose:
The `DihedralForce` class is used to calculate the forces and energies due to dihedral bonds in a molecule. There is an `DihedralForce` object present in every patch and it is responsible for calculating the dihedral interactions between local atoms as well as between local atoms and atoms from neighboring patches. This object is very similar to the `AngleForce` object with some additions to deal with the fact that dihedrals can be split between four patches, unlike angles that can only involve three patches. The `ImproperForce` class is virtually identical to this class.

Files:
`DihedralForce.h,DihedralForce.C,structures.h`

Constructor:
`DihedralForce(Patch *parentPatch, PatchList *parentList)`
The parameters `parentPatch` and `parentList` allow the object to know who owns it.

Destructor:
`~DihedralForce()`

Method of Use:

There are a small set of public routines to access this object. `initialize_timestep()` is the first of these and it simply gets the object ready for a new timestep. The function `local_init()` is used to initialize the object for calculating the local dihedral interactions. It is used during the first timestep of each cycle. The function `local_force()` is then used for all the other timesteps in a cycle to calculate the forces and energies due to local dihedrals. The function `neighbor_init()` is used to initialize the object for interactions between this patch and a neighboring patch. It builds all the data structures necessary for calculating interactions with this neighbor, and then computes the interactions. If the neighbor being initialized is one that sends all of its coordinates, then this function also returns the local indexes of atoms that need to be sent to this neighbor as bonded coordinates. The function `neighbor_force()` is used to calculate the forces between this patch and neighbor patch during mid-cycle timesteps. The function `get_energy()` returns the accumulated energy for the current timestep. While the forces due to an dihedral that is shared between patches are calculated by all of the patches, only the patch that owns the first atom in the dihedral will calculate the energy associated with this dihedral. This prevents dihedral energies from being counted more than once.

Internally, there are a few more details to what happens. The first time one of the initialization functions is called, the function `build_dihedral_list()` is used to build two lists of dihedrals. The first is a list of local dihedrals, that is, dihedrals where all four atoms are local to this patch. This list remains unchanged and is used to calculate local forces and energies. The second list is a list of all of the dihedrals that involve at least one local atom and at least one atom from another patch. This is the unassigned list. As neighbors are initialized, a list of dihedrals between the local patch and this neighbor patch is formed. This list of dihedrals is constructed by scanning the unassigned list and linking dihedrals that belong to this neighbor into the list for this neighbor. But there is a complicated case than can arise when the atoms involved in an dihedral are owned by three or four different patches. If this is the case, then the dihedral must appear in multiple neighbors lists, and space for buffering coordinates must be provided to hold the coordinates from the neighbor(s) whose message(s) arrives first. This is accomplished by storing all of the data for a particular dihedral in a structure that is then pointed to by the nodes of each neighbors linked list. This information structure may be pointed to by multiple neighbors. An illustration of this is shown in Figure 9. Once this list is built for a neighbor, the forces and energies for interactions between this patch and the neighboring patch can be determined by just walking down the list. Also, once all the neighbors have been initialized, the unassigned list should always be empty.

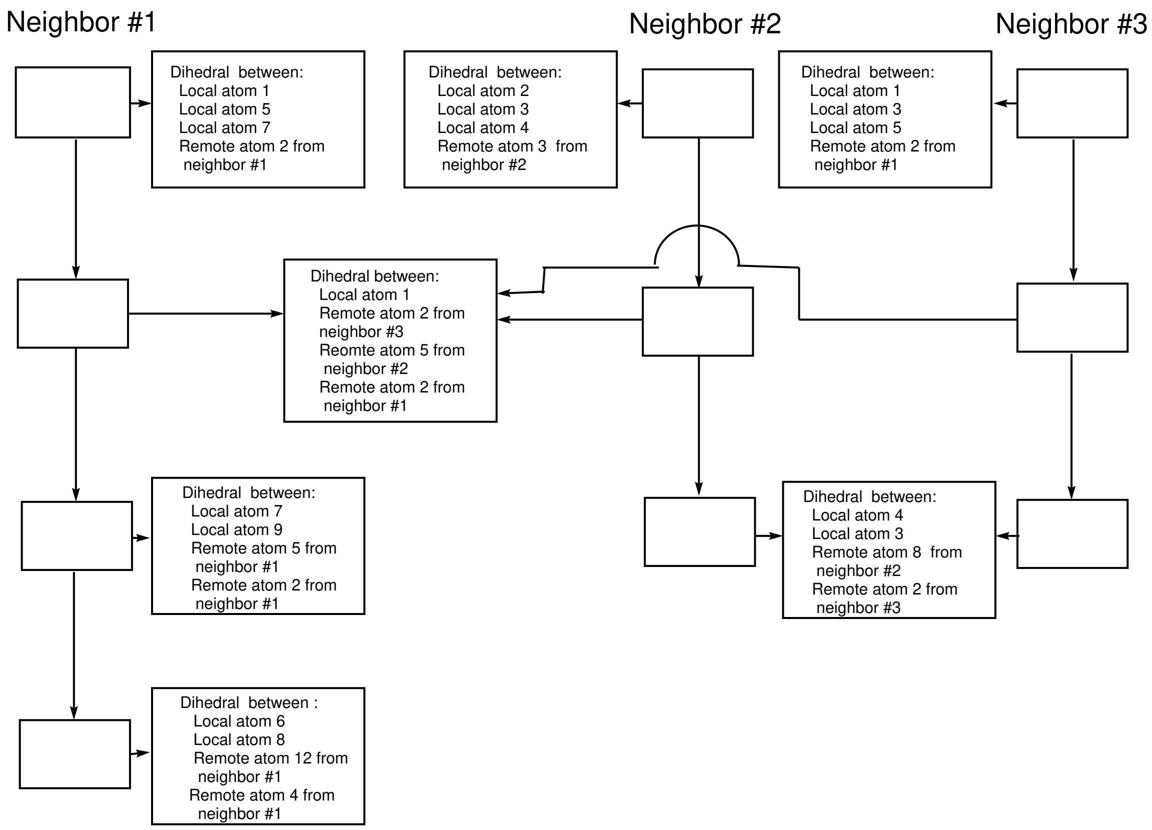

Figure 9: Example of dihedral lists with dihedrals that are shared between 1, 2, and 3 neighboring patches

During the first implementation of this object, there were design decisions that were made the should be re-examined later. The primary one is the way the list of dihedrals is constructed. Currently, this requires the maintainance of a list of all the dihedral bonds that each atom is involved in. This is fast, but ineffcient and non-scalable in memory usage. If these structure of the molecule was distributed rather than stored in total on each node as it is now, this would be more reasonable.

Functions:

- `void initialize_timestep()`
  Get the object ready for a new timestep.

- `BigReal get_energy()`
  Get the dihedral energy that has been accumulated during this timestep.

- `BigReal set_recycle()`
  No atom reassignments involved this patch, so the bond list data structures can be reused.

- `void local_init(Vector *x, Vector *f)`
  Given the local positions and force vectors for this patch, first initialize the object to perform local interactions and then calculate these interactions adding the forces to the force vector given. This function should only be used during the first timestep of a cycle.

- `void local_force(Vector *x, Vector *f)`
  Calculate the local dihedral forces for this patch given the position and force vectors. This function should be used during every timestep of a cycle except the first timestep. If this function is called before `local_init()` then bad things will happen.

- `void neighbor_init(int nid, int nAtoms, int *glbInds,`
  `Vector *localx, Vector *remotex, Vector *f, IntTree *sendtree=NULL);`

  This function initializes the data structures for calculating dihedral interactions between this patch and the neighboring patch indicated by `nid`. Once the data structures are built, then the forces and energies for these interactions are calculated and the local forces are updated. `nAtoms` is the number of coordinates received from this neighbor, `glbInds` are the global atom indexes of the coordinates received from the neighbor, `localx` are the local position vectors, `remotex` are the position vectors from the neighbor, `f` are the local force vectors to be updated, and `send_tree` is an `IntTree` object that is used to store the atom indexes of atom coordinates that need to be returned to this neighbor as bonded coordinates. If the neighbor only sends bonded coordinates, this will be `NULL`. This function should only be called once per neighbor and only during the first timestep of each cycle.

- `void neighbor_force(int nid, Vector *localx, Vector *remotex, Vector *f)`
  Calculate the dihedral interactions between the local patch the the neighbor indicated by `nid`. It does not matter whether the patch `nid` is passing us all coordinates, or only local coordinates. It does matter that initialization for this neighbor has already taken place using the routines above.

5.3.8 ElectForce

Purpose:
The `ElectForce` class is responsible for calculating all of the short range electrostatics and Van der Waals interactions. This includes all electrostatic and Van der Waals interactions within the cutoff distance. There is an `ElectForce` object present in every `Patch` object that is responsible for all the local interactions for this Patch as well as interactions between local atoms and atoms from neighboring Patches.

Files:
`ElectForce.h,ElectForce.C`

Constructor:
`ElectForce(Patch *parentPatch, PatchList *parentList)`
The parameters `parentPatch` and `parentList` let this object know about the objects that own it.

Destructor:
`~ElectForce()`

Method of Use:

The public routines to access this class follow the general interface for force objects. There is a `local_init()` routine that initializes the object for calculation of local interactions and then calculates the forces. This function is used only during the first timestep of each cycle. The function `local_force()` is then used during normal mid-cycle steps to calculate the local interactions. Similarly, the function `neighbor_init()` is used to initialize the object for interactions with a given neighbor and `neighbor_force()` is used to calculate the interaction with a given neighbor during a mid-cycle timestep. The function `get_energy()` is used to return the energy calculated during a timestep. The function `initialize_timestep()` is used to get the object ready for a new timestep.

Internally, this object relies on pairlists of interactions. A pairlist is built for the local interactions, and a pairlist is built for each neighbor that sends all of its atoms to this `Patch`. Each pairlist is built by looking at every pair of atoms involved. First, the `Molecule` class is queried to check for exclusions. Explicit exclusions, as well as bonded exclusions are checked. The bonded exclusions are applied according to the value of the `exclude` parameter given in the configuration file. If there are no exclusions, the distance between the two atoms is calculated and compared to the cutoff distance. If the distance is within the cutoff distance, then the pair needs to be calculated. At this point, the constant factor for electrostatics ($\epsilon_{14}\frac{Cq_iq_j}{\epsilon_0}$) is computed and stored in the pairlist along with the computed Van der Waals parameters A and B. From this point, the energy and forces can be computed by using the distance between the atoms and the pre-computer constants.

Functions:

- `void initialize_timestep()`
  Prepare the object for a new timestep. This basically just means setting the accumulated energy to 0.

- `void neighbor_init(int nid, int remoteNum, int *remoteGlb, Vector *remoteX, Vector *remoteF, int localNum, int localGlb, Vector *localX, Vector *localF)`
  Initialize the object for interactions with the neighboring patch specified by `nid` and then calculate the interactions with this neighbor returning the forces on the local atoms are added to the array `localF` and forces on the atoms from the neighbor are returned in the array `remoteF`. This basically involves building the pairlist, and as each pair is added to the pairlist, computing the forces and energies due to this interaction. The other parameters are: `remoteNum` is the number of atoms that the neighbor has; `remoteGlb` is the array of global atom indexes for the neighboring atoms; `remoteX` is the array of positions for the neighboring atoms; `localNum` is the number of atoms on the local `Patch`; `localGlb` is the array of global atom indexes for the local atoms; and `localX` is the array of positions for the local atoms. This function should be called only during the first timestep of each cycle.

- `void local_init(int numLocal, int *localGlb, Vector *x, Vector *f)`
  Initialize the object to calculate the local interactions and calculate them adding the forces to the array `f`. This involves building the local pairlist and calculated the forces and energies for each pair. The other parameters are: `numLocal` specifies the number of local atoms, `localGlb` is the array containing the global atom indexes for the local atoms; and `x` is the array of positions for the local atoms. This function should only be called during the first timestep of each cycle.

- `void neighbor_force(int nid, Vector *remoteX, Vector *remoteF, Vector *localX, Vector *localF)`
  Calculate the interactions with the neighbor specified by `nid`. The forces on local atoms are added to the array `localF` and the forces on neighboring atoms are returned in the array `remoteF`. The arrays `remoteF` and `localF` specify the neighboring and local positions of atoms. This function should be called during mid-cycle timesteps.

- `void local_force(Vector *x, Vector *f)`
  Calculate the local interactions during a normal timestep. The forces are added to the array `f`. The current positions of the atoms are passed in via the array `x`.

- `BigReal get_elect_energy()`
  Return the electrostatic energy calculated during the current timestep.

- `BigReal get_vdw_energy()`
  Return the van der Waals energy calculated during the current timestep.

- `void send_forces(int nAtoms, int *atoms)`
  Send the forces for DCD output.

- `void reset_pairlists()`
  This function is used to reset all the pairlist objects at the cycle boundary. This allows the pairlists to be reused without being allocated and deallocated at every cycle.

5.3.9 FMAInterface

Purpose:
The `FMAInterface` class is used as the front end for the interface to DPMTA. It is responsible for collecting the information from all the patches on a given node and passing this information to DPMTA in the format it wants. It is then responsible for getting the results from DPMTA and redistributing this information to the patches as they ask for them.

Files:
`FMAInterface.h,FMAInterface.C`

Constructor:
`FMAInterface(Bool AmIMaster)`
The `AmIMaster` flag determines whether this object is being created on the master node or not. If the object is being created on the master node, than this object is responsible for calling `PMTAinit()` to start up and initialize DPMTA.

Destructor:
`~FMAInterface()`

Method of Use:
This function has three functions for general use. The function `deposit_coords()` is used by each patch to deposit there atom indexes and coordinates. The function `execute_FMA()` is then called by the `PatchList` object to actually invoke DPMTA for the given timestep. The function `get_patch_forces()` is then called by every patch to retrieve the results of DPMTA.

Implementation:
The only tricky part of this class is keeping track of which atoms belong to which patch so that the results can be distributed back to the patches after invoking DPMTA. To maintain this information, a linked list is built during the calls to `deposit_coords()` that contains the number of atoms and the patch id number of each patch that deposits coordinates. Once all the patches have deposited their coordinates, one large array to hold all the data for this node is allocated with the data from each patch being placed sequentially into this array based on their position in the linked list. A begining index array for each patch is calculated and stored in the list. After DPMTA has been invoked, the results are redistributed to the patches by simply finding the patch id of the patch requesting the data in the list and returning the appropriate number of force values starting with that patches array index.

Functions:

- `void deposit_coords(int pid, int num, int *indexes, Vector *x)`
  Deposit the coordinates for patch number `pid`. `num` is the number of atoms being deposited, `indexes` is a list of the global atom indexes for these atoms, and `x` is an array containing the current positions of these atoms.

- `void execute_FMA()`
  This function is called by the `PatchList` class when it is time to actually invoke DPMTA.

- `void get_patch_forces(int pid, Vector *f, BigReal &patchEnergy)`
  Retrieve the results from DPMTA for patch number `pid`. The forces are returned in the array `f` and the energy for this patch is returned in `patchEnergy`.

5.3.10 FieldForce

Purpose:
This class implements the applied electric field component of the force field. It applies a constant electric field to all atoms in the simulation.

Files:
`FieldForce.h,FieldForce.C`

Constructor:
`FieldForce()`

Destructor:
`~FieldForce()`

Method of Use:
Because this is a very simple force component that requires no communication or coordinates from other patches, this class has a very simple interface. The function `initialize_timestep()` is used to prepare an object for a new timestep. The function `init()` is used during the first timestep of a cycle to compute the forces and prepare the object for the rest of the cycle. The function `force()` is used to compute forces and energies for normal mid-cycle timesteps. The function `get_energy()` is used to retrieve the energy computed during a given timestep.

Functions:

- `void init(int natoms, int *atomnums, Vector *x, Vector *f)`
  Given the local information for this patch, first initialize the object and then calculate the forces and energies adding the forces to the force vector given. This function should only be used during the first timestep of a cycle. `natoms` is the number of atoms on the current patch, `atomnums` are the global atom indexes for these atoms, `x` contains the current positions of these atoms, and `f` contains the forces for these atoms.

- `void force(Vector *x, Vector *f)`
  Calculate the forces for this patch given the position and force vectors. This function should be used during every timestep of a cycle except the first timestep. If this function is called before `init()` then bad things will happen.

- `void initialize_timestep()`
  Get the object ready for a new timestep.

- `BigReal get_energy()`
  Get the bonded energy that has been accumulated during this timestep.

5.3.11 FullDirect

Purpose:
This class provides a mechanism for directly calculating full electrostatics. This class was implemented for testing purposes, since the $O(N^2)$ cost of performing these calculations is prohibitively more expensive than the $O(N)$ means provided by DPMTA. However, it can be used for testing and performance comparisons or just to chew up extra CPU cycles if you would like.

For easy transitions between DPMTA and `FullDirect`, the interface to this class closely follows that of the `FMAInterface` class. Also, the force and energy values returned are exactly the same as those returned by DPMTA, that is, full electrostatic interactions between all atoms reguardless of distance or exclusions.

Files:
`FullDirect.h,FullDirect.C`

Constructor:
`FullDirect()`

Destructor:
`~FullDirect()`

Method of Use:

This interface to this class is meant to follow that of the `FMAInterface` class very closely so that the two can be interchanged easily. Therefore, the method of depositing cooridinates and retrieving resultant forces is identical to that used by `FMAInterface()`. The function `deposit_coords()` is used by each patch to deposit there atom indexes and coordinates. The function `get_patch_forces()` is then called by every patch to retrieve the results.

The methods needed to actually perform the calulations are different, since with DPMTA, NAMD only makes a single function call and DPMTA handles the rest. In this case, NAMD is responsible for all the message passing and computation. Therefore, there are three functions needed to perform the calculations. The function `start_direct()` is called at the begining of the first timestep of each cycle to begin the message passing and computation. The function `calc_with_node()` is used to compute interactions with a node that has sent coordinates, and the function `add_forces()` is used to add in forces calculated by another node.

Implementation:
For details on how patches interface with this class to deposit coordinates and retrieve forces, see the description of `FMAInterface`. The same mechansim is implemented here.

The calculation is performed by simple loop algorithm. If you imagine connecting all the nodes used in a loop, each node sends coordinates to the nodes half way “ahead” of it in the loop and expects to receive forces in return. Each node also expects to receive coordinates from all the nodes half way “behind” it around the loop. It will use these coordinates to compute interactions between its atoms and atoms on this node and return the forces to that node. This is just a simple way of insuring that all nodes exchange coordinates. It is not the most efficient or elegant that can be imagined, but since this class was implemented as a testing mechanism and is never expected to be used for production runs, efficiency was not the primary objective.

Functions:

- `void start_direct()`
  Called at the beginning of the first timestep of a cycle to send coordinate messages and initialize the calculations.

- `void calc_with_node(int node, Message *msg)`
  Calculate the interactions between this node and node `node`. The coordinate data for node `node` is contained in the `Message` object pointed to by `msg`.

- `void add_forces(Message *msg)`
  Add in forces calculated by another node. `msg` contains the forces due to interactions between atoms on this node and another node.

- `void deposit_coords(int pid, int num, int *indexes, Vector *x)`
  Deposit the coordinates for patch number `pid`. `num` is the number of atoms being deposited, `indexes` is a list of the global atom indexes for these atoms, and `x` is an array containing the current positions of these atoms.

- `void get_patch_forces(int pid, Vector *f, BigReal &patchEnergy)`
  Retrieve the results from DPMTA for patch number `pid`. The forces are returned in the array `f` and the energy for this patch is returned in `patchEnergy`.

5.3.12 GlobalIntegrate

Purpose:
The GlobalIntegrate class makes testing new integration schemes easier. It does this by collecting all positions, velocities, and forces in one place where they can be processed en masse, and then redistributing them to the patches for force evaluation.

Derived From:
Nothing.

Files:
`GlobalIntegrate.h, GlobalIntegrate.C`

Constructor:
`GlobalIntegrate(void)`

Destructor:
`~GlobalIntegrate(void)`

Method of Use:
Global integration is enabled by specifying the keywords "`globalTest on`" in the configuration file. Global integration is currently only available for single-processor runs and is incompatible with many features.

The user of this class generally need only be concerned with the `do_integration()` function, which is called when all of the coordinates are in place in the arrays `x, v, f` and `vh`. The default behavior is to call `do_verlet()`, a standard verlet integrator. Other experimental integrators may be present.

Functions:

- `void process_msg(int node, Message *msg)`
  Called my PatchList when a message for us comes in.
- `void deposit_atoms(int pid, int num, int *indexes, Vector *x, Vector *v, Vector *f, Vector *vh, int cycle, int first, int tstep)`
  Called by patches to deposit their coordinates and velocites.
- `void do_integration()`
  Selects and calls actual integration function.
- `void do_verlet()`
  Standard Verlet integrator (for testing).

5.3.13 ImproperForce

Purpose:
The `ImproperForce` class is used to calculate the forces and energies due to improper bonds in a molecule. There is an `ImproperForce` object present in every patch and it is responsible for calculating the improper interactions between local atoms as well as between local atoms and atoms from neighboring patches. This object is very similar to the `AngleForce` object with some additions to deal with the fact that impropers can be split between four patches, unlike angles that can only involve three patches. The `DihedralForce` class is virtually identical to this class.

Files:
`ImproperForce.h,ImproperForce.C,structures.h`

Constructor:
`ImproperForce(Patch *parentPatch, PatchList *parentList)`
The parameters `parentPatch` and `parentList` allow the object to know who owns it.

Destructor:
`~ImproperForce()`

Method of Use:

There are a small set of public routines to access this object. `initialize_timestep()` is the first of these and it simply gets the object ready for a new timestep. The function `local_init()` is used to initialize the object for calculating the local improper interactions. It is used during the first timestep of each cycle. The function `local_force()` is then used for all the other timesteps in a cycle to calculate the forces and energies due to local impropers. The function `neighbor_init()` is used to initialize the object for interactions between this patch and a neighboring patch. It builds all the data structures necessary for calculating interactions with this neighbor, and then computes the interactions. If the neighbor being initialized is one that sends all of its coordinates, then this function also returns the local indexes of atoms that need to be sent to this neighbor as bonded coordinates. The function `neighbor_force()` is used to calculate the forces between this patch and neighbor patch during mid-cycle timesteps. The function `get_energy()` returns the accumulated energy for the current timestep. While the forces due to an improper that is shared between patches are calculated by all of the patches, only the patch that owns the first atom in the improper will calculate the energy associated with this improper. This prevents improper energies from being counted more than once.

Internally, there are a few more details to what happens. The first time one of the initialization functions is called, the function `build_improper_list()` is used to build two lists of impropers. The first is a list of local impropers, that is, impropers where all four atoms are local to this patch. This list remains unchanged and is used to calculate local forces and energies. The second list is a list of all of the impropers that involve at least one local atom and at least one atom from another patch. This is the unassigned list. As neighbors are initialized, a list of impropers between the local patch and this neighbor patch is formed. This list of impropers is constructed by scanning the unassigned list and linking impropers that belong to this neighbor into the list for this neighbor. But there is a complicated case than can arise when the atoms involved in an improper are owned by three or four different patches. If this is the case, then the improper must appear in multiple neighbors lists, and space for buffering coordinates must be provided to hold the coordinates from the neighbor(s) whose message(s) arrives first. This is accomplished by storing all of the data for a particular improper in a structure that is then pointed to by the nodes of each neighbors linked list. This information structure may be pointed to by multiple neighbors. An illustration of this is shown in Figure 10. Once this list is built for a neighbor, the forces and energies for interactions between this patch and the neighboring patch can be determined by just walking down the list. Also, once all the neighbors have been initialized, the unassigned list should always be empty.

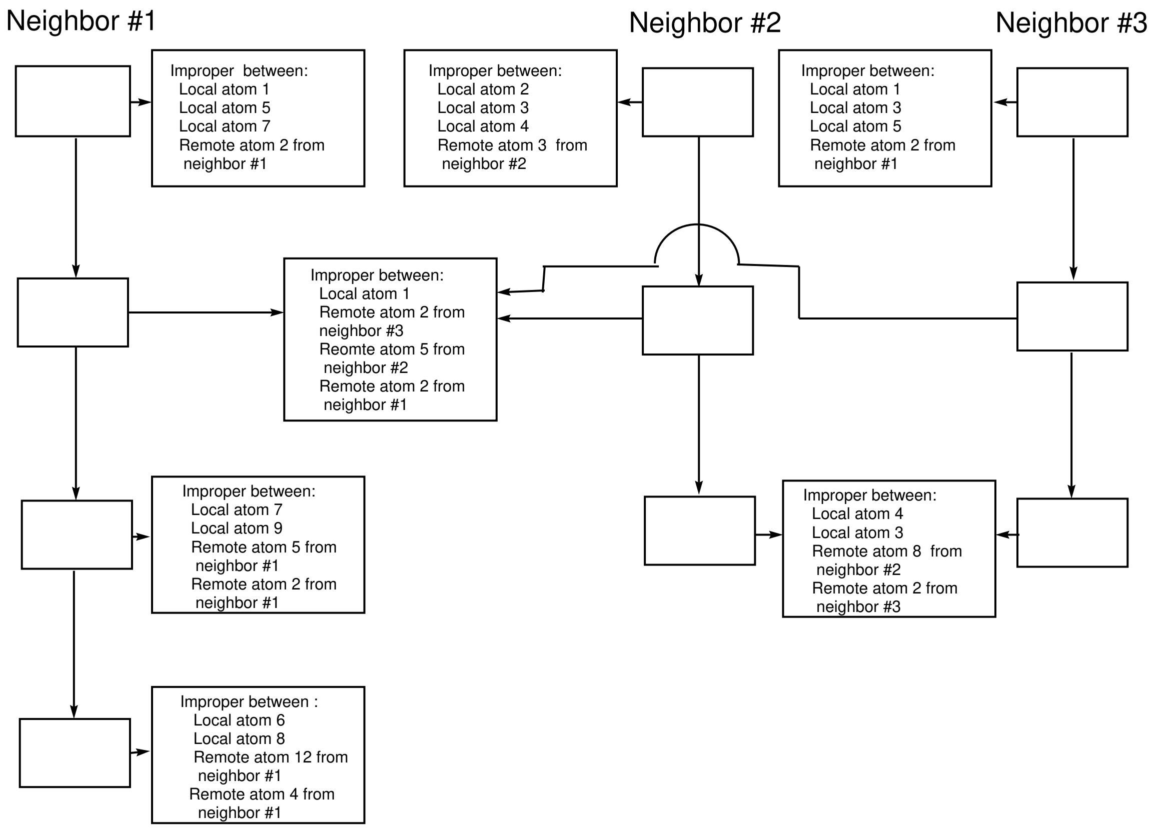

Figure 10: Example of improper lists with improperss that are shared between 1, 2, and 3 neighboring patches

For the first implementation, there are certain aspects of this object that are obviously inefficient. The most glaring of which is walking down the entire list of impropers for the entire simulation just to pick out impropers that belong to this patch. At a minimum, if this were done once per processor rather than once per patch, it would be much more efficient. If the parameters are changed so that only impropers that belong to this processor are stored on this processor, this would be much more efficient since there would be many fewer impropers to search. And also, instead of just doing a naive linear search, if the impropers were ordered in a specific way so that impropers for a specific atom could be found by using a binary search, the search time could also be reduced significantly. But for the first cut, the current algorithm is slow but effective.

Functions:

- `void initialize_timestep()`
  Get the object ready for a new timestep.

- `void set_recycle()`
  Since no atom reassignments occurred for this patch, this function tells the object to reuse the bond list.

- `BigReal get_energy()`
  Get the improper energy that has been accumulated during this timestep.

- `void local_init(int numAtoms, int *atoms, Vector *x, Vector *f)`
  Given the local positions and force vectors for this patch, first initialize the object to perform local interactions and then calculate these interactions adding the forces to the force vector given. This function should only be used during the first timestep of a cycle.

- `void local_force(Vector *x, Vector *f)`
  Calculate the local improper forces for this patch given the position and force vectors. This function should be used during every timestep of a cycle except the first timestep. If this function is called before `local_init()` then bad things will happen.

- `void neighbor_init(int nid, int nAtoms, int *glbInds,`
  `Vector *localx, Vector *remotex, Vector *f, IntTree *sendtree=NULL);`
  This function initializes the data structures for calculating improper interactions between this patch and the neighboring patch indicated by `nid`. Once the data structures are built, then the forces and energies for these interactions are calculated and the local forces are updated. `nAtoms` is the number of coordinates received from this neighbor, `glbInds` are the global atom indexes of the coordinates received from the neighbor, `localx` are the local position vectors, `remotex` are the position vectors from the neighbor, `f` are the local force vectors to be updated, and `send_tree` is an `IntTree` object that is used to store the atom indexes of atom coordinates that need to be returned to this neighbor as bonded coordinates. If the neighbor only sends bonded coordinates, this will be `NULL`. This function should only be called once per neighbor and only during the first timestep of each cycle.

- `void neighbor_force(int nid, Vector *localx, Vector *remotex, Vector *f)`
  Calculate the improper interactions between the local patch the the neighbor indicated by `nid`. It does not matter whether the patch `nid` is passing us all coordinates, or only local coordinates. It does matter that initialization for this neighbor has already taken place using the routines above.

5.3.14 Inform

Purpose:
Sends messages from the current node to a single destination node, where they are displayed. When displayed, the messages indicate the type of message (information, warning, error, or debug), the node which sent it, and the message.

An Inform object should be present on all the nodes in the machine. All but one of the nodes will just forward their messages to a single host node. The host node will periodically check for messages, and print them out.

Derived From:
Nothing.

Files:
`Inform.h, Inform.C`

Constructor:
`Inform(char *name)`
where `name` is the name for this particular Inform instantiation. This name is displayed with the node number and message when the message is printed out.

Destructor:
`~Inform(void)`

Method of Use:
Each Inform object has an *active message buffer*. To put data in this buffer, use the << operator. For example:

```
inform << "Example message number " << 2 << "/n Second line." << sendmsg;
```

The << operator will accept strings and all simple data types, i.e char, int, double, etc.

The `sendmsg` manipulator can be used to signal that the message is complete and should be sent. Its use is equivalent to doing the following:

```
inform << "Example message number " << 2 << "/n Second line.";
inform.send();
```

The `level1`–`level10` manipulators allow message levels to be associated with messages, so the output can be limited with the `output_level()` functions.

Newline characters in the message are used to break up the message into multiple lines when the message is printed out. When displayed, each line up to a newline is printed with a leading message. The above examples, when executed on node 2 by an Inform object "Info", generate the following messages on the host node:

```
Node 2:Info> Example message number 2
Node 2:Info> Second line.
```

Functions:

- `use_comm(Communicate *c, int newnode = 0, int newtag = 0)`
  This provides the Inform object with data needed to send messages. The given Communicate object is used to send messages to the destination node `newnode`, with the given tag `newtag`. This can be called at any time. By default, an Inform object has no Communicate object, and will send to node 0 with tag 0. If no Communicate object has been provided when a message is told to be sent, the message is printed to the console.

- `void on(int) || int on(void)`
  Turns on or off, or queries, whether the Inform message will send or display it's messages. By default, messages are sent.

- `int output_level() || Inform& output_level(int ol)`
  Sets or retrieves the current output level. Each output message has a message level associated with it, and the output level prevents messages with a message level greater than the output level from being printed.

- `int msg_level() || Inform& msg_level(int ol)`
  Sets or retrieves the default message level. Messages levels can be set for individual messages using the `level[1-10]` manipulators.

- `void destination(ostream *)`
  For the host node, set the destination ostream object to send the messages. By default, this is cout, but can be changed to anything, for example a log file.

- `int check(void)`
  On the host node, this checks for any incoming Inform messages from other nodes, and prints them out to the destination ostream. This must be called periodically to keep the incoming messages from accruing. For non-host nodes, this does nothing. The number of messages received during this latest check is returned.

- `int send(void)`
  An Inform object has a *current message buffer*, to which data is added by using the << operator (see below). Once a message has been set up, the send routine takes the current message and forwards it to the host node. On the host node, this results in the message being printed out immediately.

5.3.15 IntList

Purpose:
The `IntList` class provides a flexible list of integers with constant time access to any element and expandable storage that is transparent to the user. It is basically an array of integers where all the details are maintained internally.

Files:
`IntList.h`,`IntList.C`

Constructor:
`IntList(void)`

Destructor:
`~IntList()`

Operators:

- `[]`
  The bracket operators can be used to access an element in the `IntList` in constant time just as if it were an array.

Functions:

- `int sort()`
  Sorts the list.
- `int num()`
  Return the number of integers in the list.
- `void add(int newint)`
  Add a new integer to the list.
- `void find(int target)`
  Uses a binary search to find the index of the indictated target element.
- `void merge(int match)`
  Adds a new value only if it doesn't already exist. It assumes the list is already sorted, and sorts it when it is done.
- `void merge(IntList &il)`
  Adds all the contents of another IntList to this Intlist, and sorts when done.
- `int hasany(int n, IntList &il)`
  Looks for n or more common values between two sorted lists.
- `int hasexactly(int n, IntList &il)`
  Looks for exactly n common values between two sorted lists.

- `void clear()`
  Removes all elements from the list.
- `int excise(int *indices, int how_many)`
  Removes specified elements from list.

5.3.16 IntTree

Purpose:
The `IntTree` class is a specialized binary tree of integers. It could easily be changed for more general use, but at the moment its sole purpose in being is to keep track of atoms that need to be sent as bonded coordinates. As such, `IntTree` takes a series of integers as input. It then stores only the distinct values as a binary tree. Upon request, all the values in the tree can be returned in the form of an array of integers.

The `IntTree` class is used to track atoms that need to be sent to a neighbor as bonded coordinates. A single `IntTree` object is used and is passed to each of the bond force objects as the initialize themselves with a neighbor who sends all of their coordinates. Each object places the local atom indexes of the atom coordinates that need to be sent to the neighbor into the `IntTree` object. Since the tree only stores distinct values, after the object has been passed to all the force objects the contents can be dumped to an array of integers that then represents all the atoms that need to be sent to this neighbor.

Derived From:
Nothing.

Files:
`IntTree.h,IntTree.C`

Constructor:
`IntTree()`

Destructor:
`~IntTree(void)`

Method of Use:
There are only three public functions for this class. `size()` returns the number of nodes in the tree, `add_value()` adds a value to the tree, and `make_array()` converts the contents of the tree into an array of integers.

Functions:

- `int size()`
  Return the number of nodes in the tree.
- `void add_value(int intval)`
  Add a value to the tree. Since only distinct values are stored, if the value already exists in the tree, the tree remains unchanged.
- `int *make_array()`
  Convert the values in the tree into an array of integers. If there are currently no values in the tree, then `NULL` is returned.

5.3.17 Integrate

Purpose:
The Integrate class performs the integration to calculate the new positions of atoms belonging to the parent patch. Every patch has an instance of Integrate class. The current implementation uses the velocity form of the Verlet-I or Leapfrog method.

Derived From:
Nothing.

Files:
`Integrate.h, Integrate.C`

Constructor:
`Integrate(void)`: New empty integrator, with no intermediate velocities.

`Integrate(int numAtoms, Message *msg)`: An integrator is created to contain `numAtoms`. Any state is extracted from the message, which is not freed after use.

Destructor:
`~Integrate(void)`

Method of Use:
Every patch creates an instance of Integrate class and uses the same integrator for its lifetime. The integrator is initialized by invoking `init()` and supplying the number of local atoms maintained by the patch. There are three more functions that are invoked by the patch: `do_integration()` performs the integration and updates the coordinates and velocity arrays owned by the patch. When atoms move from/to patches, the `add_atoms` and `delete_atoms()` functions are used to adjust the integrator.

Functions:

- `void init(int)`
  Initialize the integrator object.
- `void send_atom_info(Message *msg, int numAtoms, LintList *send_atoms)`
  Adds any internal state information for the atoms in `send_atoms` to the message for atom redistribution.
- `void prepare_to_receive_atoms(int numAdded, int numRemoved, LintList atomsRemoved)`
  This function reallocates the necessary data structures during atom reassignment. It allocates a new array for the half-step velocities, copies over old information that is to be preserved, and gets ready to receive new information that will be coming.
- `void receive_atom_info(int nAtoms, Message *msg)`
  This function takes a message from a neighboring patch containing information about atoms

that have migrated to this patch. It then uses this information to add the atoms its local list.

- `void do_integration(Vector *x, Vector *v, Vector *f, int *global_nums, int timestep, int first_time_step)`
  Perform integration using the selected integration scheme. For some integration schemes, the integrator maintains an intermediate velocity vector (from time step $n - \frac{1}{2}$).

- `void start_global(Vector *x, Vector *v, Vector *f, int *global_nums, int new_cycle, int first_time_step, int timestep)`
  Starts integration using the GlobalIntegrate object.

- `void get_new_velocities(Vector *x, Vector *v, Vector *f, int *global_nums, int first_time_step)`
  Performs a partial integration to determine velocities, but positions are not updated. This is useful for temperature rescaling.

- `void save_state(Message *msg)`
  Saves the integrator state into a message for re-creation after patch migration.

Derived Classes:
None.

5.3.18 LintList

Purpose:
The `LintList` class is a simple, completely inlined implementation of a linked list of integers. It is not meant for nor does it provide anything fancy, just the ability to add elements to the list and to traverse the list.

Files:
`LintList.h`

Defined Constants:
`LIST_EMPTY` - This is the value is returned when the end of the list is encountered.

Constructor:
`LintList()`

Destructor:
`~LintList()`

Method of Use:
There are only three functions used to manipulate this class. The function `add()` is used to add a new integer to the list. Additions are accomplished in constant time. The function `head()` returns the first value in the list and sets the current position in the list to be the head of the list. The function `next()` returns the next value in the list and moves the current position in the list to this element. Both `head()` and `next()` return `LIST_EMPTY` if the list is completely empty, or if the end of the list has been reached.

So to traverse the list, `head()` is called once followed by repeated calls to `next()` until the value `LIST_EMPTY` is returned.

Functions:

- `int head()`
  Returns the first value of the list and set the current position in the list to the head of the list. If the list is currently empty, `LIST_EMPTY` is returned.
- `int next()`
  Returns the next value of the list and sets the current position in the list to this item. If the end of the list has been reached, `LIST_EMPTY` is returned.
- `void add(int addvalue)`
  Adds a value to the list. Items are always added to the tail of the list in constant time.
- `int num()`
  Returns the number of items in the list in $O(n)$ time.
- `int num(int match)`
  Returns the number of occurrences of `match` in the list in $O(n)$ time.

- `void del(int match)`
  Deletes the first occurrence of `match` from the list.

- `void del(LintList &match_list)`
  Deletes the first occurrence of each item in `match_list` from the list.

- `void merge(int merge_item)`
  Adds the new item only if it is not already in the list.

- `void merge(LintList &merge_list)`
  Adds each item in the `merge_list` only if it is not already in the list.

5.3.19 LoadBalance

Purpose:
The `LoadBalance` object is responsible for determining a list of patches to be moved to new nodes. It currently does this by collecting all of the load information collected in the `LoadStats` structure on the master node. The master node then analyzes this information, and creates a message for each node. This message contains the number of new patches to receive, and the patch numbers of patches to send to new nodes. The patch migration is actually performed by the `Node` and `Patch` objects. Since this module is not used for production runs, and will be extensively modified, the following description will be brief.

Files:
`LoadBalance.h, LoadBalance.C`

Constructor:
`LoadBalance(int myNode)`

Destructor:
`~LoadBalance()`

Method of Use:
At the end of each timestep, `deposit_load_stats` is called to accumulate all the timer values recorded during the timestep. At the end of some timestep shortly before a load balance cycle, `send_load_info` sends these accumulated values to the master node. The master node calls `receive_load_info` to analyze these values, and creates a message with tag `LDSTATINFOTAG` which tells each processor how many new patches to receive, which patches to send away, and the destination node for each departing patch. This message and the coordination of the patch migration is handled by the `PatchList` object on each node.

Functions:

- `void deposit_load_stats(LoadStats *load_stats)`
  Accumulate the load for the current timestep with the local total statistics.
- `void send_load_info(void)`
  Send the load statistics collected by `deposit_load_stats` to the master node in a `PATCHLOADTAG` message.
- `void receive_load_info(Message *load_message)`
  Whenever a `PATCHLOADTAG` message is received by the master node, this function is called to process it. The function expects one message from each node, and the new patch assignments are determined when the last message has been received.
- `void patch_changes_complete(void)`
  This function is called when patch migration is complete, so the object can start accumulating new load data.

5.3.20 LongForce

Purpose:
This object provides the patch level interface to the electrostatics module. It provides the means to deposit patch data to the node level FMAInterface object and to return the forces to the Patch for use in the computations. In NAMD, the electrostatic forces acting on an atom is divided into two groups: short range and long range forces. A full electrostatic calculation module (such as DPMTA or FullElect) returns the total electrostatic forces for each atom. LongForce module extracts the forces due to short range (interactions within the cutoff) and due to interactions between excluded pairs of atoms from the total forces. The long range forces are used during the cycle without recomputing them. However, short range forces are recalculated at every time step within a cycle. This module depends also on the multi-time stepping scheme. Some schemes such as Verlet-X require previous forces. The application of long-range forces depends on the multi-time stepping scheme too. For details, please refer to the code (particularly the function `get_long_force()` and the section 3.8.

Derived From:
Nothing.

Files:
`LongForce.h LongForce.C`

Constructor:
`LongForce(int pid)`: Initializes the `LongForce` object. `pid` is the patch identifier (of the owner of the LongForce object).
`LongForce(int pid, int n, Message *msg)`: reinitializes the `LongForce` object when the patch `pid` has moved (migrated) to another processor. The state of the object is restored from `msg`.

Destructor:
`~LongForce(void)` : frees allocated memory

Method of Use:
The interface functions of this object can be classified into three groups: (a) initialization , (b) calculation of long forces and providing access to them, and (c) handling atom redistribution (because this object has internal information per atom. If an atom moves to another patch, the related atom based information from the `LongForce` object must be communicated to the instance of the `LongForce` object of the destination patch). The description of the functions are given below:

Functions:

- `void initialize_timestep(int step)`
  sets the timestep value.
- `void end_cycle()`
  resets the internal data structure for the next cycle.

- `void init(int n)`
  this function is called when the `LongForce` object is created. It initializes the internal data structure that depends on the number of atoms `n` in the patch.

- `void initialize_first_timestep(int n, int *localGlb, Vector *x, int s)`
  this function deposits the coordinates `x` of the `n` atoms of the owner patch to the full-electrostatic module (such as DPMTA) at the beginning of each cycle.

- `void calc_eff_force(Vector *subForce, BigReal subEnergy)`
  this function calculates the long range forces. It first gets the full-electrostatic forces (the computation of the full-electrostatic forces is triggered by `init_first_timestep`) Then, the long range forces are calculated by substracting the forces due to the short range and excluded interactions `subForce`.

- `void get_long_force(Vector *localF)`
  this function adds the long range forces to the forces kept by the patch `localF`. The addition, however, depends on the multi-time stepping scheme is used. In the NAIVE case, long forces are added at each time step. For VERLET I scheme, however, long forces are multiplied by the length of the cycle and added once at the beginning of the cycle. During the intermediate time steps long range forces will not be added in VERLET I. For details please see the code.

- `void send_atom_info(Message *msg , int numSend, LintList *send_atoms)`
  this function prepares the information to be transferred to another `LongForce` object for atoms that are leaving.

- `void prepare_to_receive_atoms(int numAdded, int numRemoved, LintList atomsRemoved)`
  this function receives information about the changes due to atoms moving in or out of the patch. It rearranges internal data structure to accomodate the new information which will be received soon.

- `void receive_atom_info(int nAtoms, Message *msg)`
  this function gets the information `msg` sent by a neighboring patch due to atoms moving in.

- `void send_forces(int nAtoms, int *atoms)`
  this function deposits the long range forces to the `Collect` object so that they can be gathered and output in a force DCD file.

- `void save_state(Message *msg)`
  this function saves the state of the `LongForce` object into `msg` for patch migration.

- `void BigReal get_energy()`
  returns the energy due to long_forces.

5.3.21 Message

Purpose:
Encapsulates data of various types that is to be sent to another node in a parallel environment. A Message object is created, filled with the data to be sent, and given to a Communicate object to be delivered. Each Message consists of one or more *items*, of different types. A user can query how many items there are, the type and size of each item, and retrieve the items.

Derived From:
Nothing.

Files:
`Message.h, Message.C`

Constructor:
`Message(void)`

Destructor:
`~Message(void)`

Enumerations:

- `Types :  CHAR, SHORT, INT, LONG, FLOAT, DOUBLE, UNKNOWN`

Method of Use:
A Message object is used either to send data to another node, or to retrieve data that has been received from another node.

To send a Message, first a new Message object is created, and then data is put into the Message using the `put` function. For example:

```
Message *msg = new Message;
msg->put("Start of message.").put(5,intarray).put(3.1415);
communicate->send(msg, node, tag);
```

The `put` routine is overloaded to accept all basic data types, either a single value (one argument), or an array (two arguments, number of elements and pointer to array).

There are two ways to store the data for each item in the Message: make a copy of the data, or store a pointer to the data. By default, data you provide in a call to `put` is copied into some new storage area allocated by the Message object. For scalar data, a copy of the data is always made. However, for arrays of data, you can instead choose to just have the Message store the pointer to the data that you provide in the `put` function call. Doing this eliminates the need to allocate new memory and copy the data, but requires you to make sure the data is still available when the Message is actually sent to the destination node (which could be some time after the creation of the Message).

When just a pointer is stored, the Message object by default will not free up that storage when the Message is deleted, as it normally does for data it copies to its own storage. You can also choose to have the storage space for data you `put` be freed up by the Message object when it is deleted. This might be advantageous if a Message is being constructed with data you have allocated earlier, but have no further need of other than sending out.

When calling `put` for arrays of data, there are two optional parameters:

1. An optional third argument which indicates whether to *copy* the data (TRUE) or just *reference* the data via the given pointer (FALSE). This is TRUE by default (i.e. by default data is copied).

2. An optional fourth argument which indicates whether to delete the storage spaced used by the Message for the data (TRUE) or just leave the storage space unchanged (FALSE). This option is ignored if the third argument is TRUE, that is when a copy is made, the storage is always deleted (since it is allocated by the Message object in the first place). This is FALSE by default (i.e. by default if data is not copied, the storage will not be freed up).

For versions of the `put` routine for scalar data, there is only one argument (the data). There is no optional second or third argument.

For example, the command given just above copies the data from `intarray`, while this example:

```
msg->put(5,intarray,FALSE,TRUE);
communicate->send(msg, node, tag);
```

just specifies for `msg` to store the pointer `intarray`. When the Message is deleted, the memory pointed to by `intarray` is freed up for future use.

There are two ways to get data from a Message:

1. Basic loop. The `items` routine will report how many data items there are. The `type(n)` and `size(n)` routine report the type and size (in number of elements) for each item, and each item can be retrieved via the `item(n)` routine.

2. Iteration. Each Message object has a *current* item. The `reset()` routine sets the current item to the beginning of the Message. After this has been done, the `type()`, `size()`, and `item()` routines with no arguments will return data about the current item. To retrieve data and automatically move the current item to the next in the list, use the `get` routine. This will retrieve the data in the current item, place it in the storage given by the argument, and increment the current item.

Note that new items are appended to the end of a Message. It is possible to receive a Message, get some data from it, add new items to it, and forward it on to another node. It is also possible to delete individual items from the message (using the `del(n)` routine), or delete all items in the message (with the `clear()` routine).

Functions:

- `ostream &operator<<`
  The `<<` operator can be used on an ostream object (such as `cout` or `cerr`) to print a summary of the current message. For example:

```
Message msg;
cout << "Contents of the message:  " << msg;
```

- Message &clear(void)
  Deletes the storage for ALL the items in the current message.

- int items(void)
  Returns the number of items currently stored in the Message. Items are numbered 0 ... items() - 1.

- int size(int n = (-1))
  Returns the size (in number of elements) of the nth item. If no argument is given, returns the size of the current item.

- Types type(int n = (-1))
  Returns the type (as one of the enumeration Types listed above) of the nth item. If no argument is given, returns the type of the current item.

- void *item(int n = (-1))
  Returns a pointer to the nth item data. Since data can be of many types, returns a pointer to void. If no argument is given, returns a pointer to the current item data.

- Message &del(int n = (-1))
  Deletes the nth item data storage. If no argument is given, does so for the current item.

- Message &reset(void)
  Resets the current item to be the first one in the message.

- Message &skip(void)
  Moves the current item on to the next item in the message.

- Message &back(void)
  Moves the current item back to the previous item in the message.

- Message &current(int n)
  Sets the current item to be the nth item in the message.

- Message &put(char *d, int copy = TRUE, int delstor = FALSE)

- Message &put(<type> d)

- Message &put(int n, <type> *d, int copy = TRUE, int delstor = FALSE)
  The put routine adds new items to the end of the Message. The first form adds a null-terminated string, the second a scalar of type <type>, and the third an array of type <type>, size n. The third form has two optional arguments described above. It returns a Message reference, so several put calls may be strung together as shown in the example above.

- Message &get(<type> &d)

- `Message &get(<type> *d)`
  The get routine retrieves the data from the current item, and copies it into the argument d. The caller must know what type of data is to be retrieved, and call get with the appropriate argument type. Space must be provided by the caller to store the retrieved data, it is not allocated by get. The first form retrieves a scalar value, the second a vector. If the type of the current item does not match the argument, or there is no current message, get does nothing. The size and type of the current message can be found via size() and type(), respectively.

5.3.22 MessageManager

Purpose:
The `MessageManager` class is a completely inlined class that provides an efficient storage mechanism for Messages that have been received.

This class allows the retrieval of a message with a specific tag from any node in constant time. This is the case used to retrieve a virtually all messages in NAMD. Searches for messages with any tag but from a specific node or from any node with any tag may require $O(N_{msgs})$ time where N_{msgs} is the total number of messages currently stored. (There are currently no such searches done in NAMD.)

Implementation:
This class is currently implemented using an array of `MessageQueue` objects, where each queue stores messages with a specific tag. An array of `MessageQueue` objects is maintained where there is one object for each tag used in the program. Therefore all messages with a given tag are stored in a single `MessageQueue` object.

This provides a means for constant time retreival for messages with a specific tag, which is what is desired based on the design of NAMD. However, please note that this implementation sacrifices some flexibility for this speed. One, all the message tags used must be in a continguous range so that the mapping of tags to `MessageQueue` objects can be maintained. Next, the maximum tag value must always be clearly defined, otherwise nasty things happen. This restrictions are implemented and documented in the file `common.h` where all the tags are defined. These restrictions seem quite reasonable given the performance improvements and the design of NAMD, but this may be changed in the future or if this class is stolen for other purposes.

Two of `MessageManager` objects are used by each `Communicate` class. one to store messages from other nodes and another to store messages sent by patches on the same node.

Files:
`MessageManager.h`,`MessageManager.C`

Constructor:
`MessageManager()`

Destructor:
`~MessageManager()`

Functions:

- `int num()`
  Return the number of Messages currently stored in the object.
- `void add_msg(MsgList *new_msg)`
  Add the Message enclosed in the structure `new_msg` into the object. This addition is performed in constant time.
- `MsgList *find_msg(int &tag, int &node)`
  Find the first message with the tag and node specified by `tag` and `node`. If `tag` is -1, the first

message matching the `node` requirement with any tag is returned. If `node` is -1, then the first message matching the `tag` requirement from any node is returned. The running time for the various types of searches are detailed above.

5.3.23 MessageQueue

Purpose:
The `MessageQueue` class is a almost completely inlined class that provides a FIFO queue of `Message` objects. The properties of these queues are constant time addition of messages and retrieval of the head of the queue. Searches of the queue for messages with a specific property (such as Node or Tag value) are $O(N)$ where N is the number of messages stored in the queue.

This class is currently implemented as a doubly linked list.

The `MessageQueue` class is currently only used by the `MessageManager` which uses these queues to efficiently serach for stored messages.

Files:
`MessageQueue.h`,`MessageQueue.C`

Constructor:
`MessageQueue()`

Destructor:
`~MessageQueue()`

Functions:

- `int num()`
  Return the number of Messages currently stored in the Queue.
- `void add_msg(MsgList *new_msg)`
  Add the Message enclosed in the structure `new_msg` into the queue. This addition is performed in constant time.
- `MsgList *get_head(int &tag, int &node)`
  Return a pointer to the Message at the head of the queue and take it off the queue. If the list is empty, `NULL` is returned. This operation is also performed in constant time.
- `MsgList *get_msg_by_tag(int &tag, int &node)`
  Return a pointer to the first message on the queue with the tag indicated by `tag` and remove it from the queue. If there is no message on the queue with a tag that matches, `NULL` is returned. This operation is accomplished in $O(N)$ time. In the current implementation of `MessageManager`, this operation is never used. But it is provided for completeness.
- `MsgList *get_msg_by_node(int &tag, int &node)`
  Return a pointer to the first message on the queue with the node indicated by `node` and remove it from the queue. If there is no message on the queue with a node that matches, `NULL` is returned. This operation is accomplished in $O(N)$ time.

5.3.24 Molecule

Purpose:
The Molecule class is used to read in, store, and access the molecular structure read in from the XPLOR style `.psf` file. This information includes a list of all the atoms along with their mass and charge, a list of all bonds, angles, dihedrals, and impropers, and a list of the explicit electrostatic exceptions. A Molecule object will reside on each of the processors. The `.psf` file is read in on the master node and the information is then be distributed to the other processors. The routines needed to access this data during the simulation have not currently been implemented.

Derived From:
Nothing.

Files:
`Molecule.h, Molecule.C, structures.h`

Constructor:
`Molecule()`

Destructor:
`~Molecule(void)`

Method of Use:
The function `read_psf_file(char *name, Parameters *params)` is used to read in the parameter file specified. This called should be used on the master process to read in the psf file specified in the configuration file. This information will later be distributed to all of the other processors. The Parameters object specified by `params` is used to verify that all of the parameters necessary have been read in when the `.psf` file is read.

The public variables num<structure_type> specify the number of each type of structural information has been read in from the `.psf` file. The routines print_<structure_type>() are provided to print out the information read in from the `.psf` file for debugging purposes.

Routines to access this data during the simulation have currently not been implemented.

Public variables:

- `numAtoms`
- `numBonds`
- `numAngles`
- `numDihedrals`
- `numImpropers`
- `numExclusions`

- numConstraints
- numMultipleDihedrals
- numMultipleImpropers

Functions:

- `void read_psf_file(char *name, Parameters *params)`
  Read in the `.psf` file specified by `name` and use the Parameters object `params` to verify that all of the parameters necessary for this structure have been specified.
- `void send_Molecule(Communicate *comm)`
  Send the molecule structure from the master to the clients.
- `void receive_Molecule(Message *msg)`
  Initialize a client Molecule object from the message sent by the master Molecule object.
- `void build_constraint_params(StringList *consref, StringList *conskfile, StringList *conskcol, PDB *initial_pdb, char *cwd)` This function builds all the parameters that are necessary to do harmonic constraints. This involves looking through one or more PDB objects to determine which atoms are constrained, and what the force constant and reference position is force each atom that is constrained.
- `void build_langevin_params(StringList *langfile, StringList *langcol, PDB *initial_pdb, char *cwd)`
  This function builds the array of b values necessary for Langevin dynamics. It takes the name of the PDB file and the column in the PDB file that contains the b values. It then builds the array langevinParams for use during the program.
- `Real atommass(int anum)`
  Gets the mass of an atom.
- `Real atomcharge(int anum)`
  Gets the charge of an atom.
- `Index atomvdwtype(int anum)`
  Gets the van der Waal type of an atom.
- `Bond *get_bond(int bnum)`
  Retrieve a bond structure.
- `Angle *get_angle(int anum)`
  Retrieve an angle structure.
- `Improper *get_improper(int inum)`
  Retrieve an improper structure.
- `Dihedral *get_dihedral(int dnum)`
  Retrieve a dihedral structure.

- `const char *get_atomtype(int anum)`
  Retrieve an atom type.
- `LintList *get_bonds_for_atom(int anum)`
  Gets the list of bond numbers for the given atom.
- `LintList *get_angles_for_atom(int anum)`
  Gets the list of angle numbers for the given atom.
- `LintList *get_dihedrals_for_atom(int anum)`
  Gets the list of dihedral numbers for the given atom.
- `LintList *get_impropers_for_atom(int anum)`
  Gets the list of improper numbers for the given atom.
- `Bool checkexcl(int atom1, int atom2)`
  Check for an explicit or bonded exclusion between these two atoms.
- `Bool check14excl(int atom1, int atom2)`
  Check for a 1-4 exclusion between these two atoms. This function is only valid when the exclusion policy is set to scaled1-4.
- `Bool is_atom_constrained(int atomnum)`
  Return TRUE if the specified atom is constrained.
- `void get_cons_params(Real &k, Vector &refPos, int atomnum)`
  Gets the harmonic constraint parameters for the specified atom.
- `Real langevin_param(int atomnum)`
  Gets the Langevin parameter for the specified atom.
- `Real langevin_force_val(int atomnum)`
  Gets the Langevin force value for the specified atom.
- `void print_atoms()`

- `void print_atoms()`
  Print out a list of all the atoms read from the `.psf` file.
- `void print_bonds()`
  Print out a list of all the bonds read from the `.psf` file.
- `void print_exclusions()`
  Print out a list of all the explicit exclusions read from the `.psf` file.

5.3.25 Node

Purpose:
The `Node` class is where all of the actual work for the simulation is done. It can be thought of as the `main()` routine for each processor to run during the simulation. It contains all of the other objects to be used during the simulation and the highest level control structure for the simulation. One `Node` object will exist on each processor and will handle all of the activity for the simulation on this processor.

Derived From:
Nothing.

Files:
`Node.C, Node.h`

Constructor:
`Node(int id)` – `id` specifies what processor this node resides on

Destructor:
`~Node(void)`

Method of Use:
Once the `Node` object is created on each processor, there are basically only two routines that are called externally on each node. One performs all of the necessary setup and is different for the master node and the client nodes. On the master node, the routine used is `master_startup()` and on the client node `client_startup()` is used. Once the `Node` object is started, the `doSimulation()` function is called, and the `Node` object then takes over the control of the simulation.

Objects contained within the Node:
The purpose of the `Node` object is to hold all of the other objects in the program and control their execution. The objects that are contained by each `Node` object are:

- `Molecule *structure` – molecular structure from the .psf file
- `Parameters *params` – energy parameters from parameter file(s)
- `SimParameters *simParams` – global simulation parameters such as timestep size, number of timesteps, etc.
- `PatchDistrib *patchMap` – mapping of patches to processors
- `PatchList *patchList` – object which contains all of the `Patch`s that are assigned to this processor
- `Collect *collect` – object which gathers information periodically at the master node for output and energy computation.
- `PDB *pdbData` – Data object produced from the PDB file, only used on the master node.

- `Output *output` – Performs status and file output.
- `FullDirect *fullDirect` – Direct electrostatic calculation object.
- `GlobalIntegrate *globalIntegrate` – Object which performs global integration on the master node.
- `FMAInterface *FMA` – FMA (DPMTA) calculation interface object.
- `LoadBalance *loadBalancer` – Load monitoring/management object.

Public variables:

- `com` – center of mass of all simulated atoms.
- `myNode` – Which node am I.

Functions:

- `void velocities_from_PDB(Vector *v, char *fname)`
  This function is a private function used by the master node to determine the initial velocities for all the atoms from a PDB file. The name of the PDB file to read is specifed by `fname` and the arrays of Vectors to populate is given by `v`.
- `void random_velocities(Vector *v, BigReal temp)`
  This is a private function used by the master node to determine the initial velocities for all the atoms using a random Maxwell distribution. The temperature to achieve with this distribution is given by the parameter `temp` and the array of Vectors to populate with the velocities is given by `v`.
- `void master_startup(int argc, char **argv)`
  Initialize the `Node` object on the master node. This involves starting up all the other objects, reading in the appropriate files, and then sending this information to all of the client nodes. The arguments `argc` and `argv` are used to obtain the configuration file name that is passed on the command line.
- `void client_startup()`
  Initialize the `Node` object on a client node. This involves allocating all of the objects and then receiving the information for each one from the master process.
- `void doSimulation()`
  This function contains the logic with controls the simulation.
- `void deposit_temperature(int timestep, Real temp)`
  Gathers temperatures for temperature rescaling.
- `BigReal get_rescale_factor(int timestep)`
  Retrieve the rescaling factor for temperature rescaling.

- `BigReal get_last_temperature(int timestep)`
  Get the last temperature for temperature coupling.

- `void read_binary_coors(char *filename, PDB *pdbobj)`
  Reads the initial coordinates from a binary restart file.

5.3.26 Output

Purpose:
The `Output` class is used to produce the useful output of NAMD such as the energy values for a timestep, DCD trajectory files, restart files, and the final position and velocity files.

This object exists only on the Master Node and is owned and accessed by the `Collect` object on the Master Node.

Files:
`Output.h Output.C dcdlib.h dcdlib.C`

Constructor:
`Output()`

Destructor:
`~Output()`

Method of Use:
There are only three public functions in the `Output` class. They are used by the `Collect` object to pass along values that it has collected. The `Output` object then takes this data and calls the appropriate private routines to actually output the data.

The three public routines are `energy()`, `coordinate()`, and `velocity()`. The purpose of each routine is pretty self-explanatory.

Private functions:

- `void output_dcdfile(int timestep, int n, Vector *coor)`
  This function is used to write out the trajectory file in binary DCD format. It serves as an interface to the routines provided in `dcdlib` that do the actually formatting and writing to the DCD file. `timestep` indicates the timestep the coordinates being written are for, `n` indicates the number of coordinates being written, and `coor` is an array of vectors that contains the actual coordinates to be written.

- `void output_restart_coordinates(Vector *coor, int timestep)`
  Writes out the coordinates passed in `coor` for timestep `timestep` to the restart file specifed in the configuration file using the option `restartname` in PDB format.

- `void output_restart_velocities(Vector *vel, int timestep)`
  Writes out the velocities passed in `vel` for timestep `timestep` to the restart file specifed in the configuration file using the option `restartname` in PDB format.

- `void output_final_coordinate(Vector *coor, int timestep)`
  Writes out the coordinates passed in `coor` for timestep `timestep` to the final output file specifed in the configuration file using the option `outputname` in PDB format.

- `void output_final_velocities(Vector *vel, int timestep)`
  Writes out the velocities passed in `vel` for timestep `timestep` to the final output file specifed in the configuration file using the option `outputname` in PDB format.

Functions:

- `void energy(int timestep, BigReal *energies)`
  This function takes the energies for the timestep specified by `timestep` in the array `energies` and outputs them, currently just to the screen. It also sums all the component energy to get a total energy sum and computes and reports the temperature based on the value of the kinetic energy.
- `void coordinate(int timestep, int n, Vector *coor)`
  Takes the positions for timestep `timestep` from the array `coor` and calls the appropriate output routines. `n` is used to specify the number of coordinates that are passed in. At the moment, the output routines called include `output_dcdfile()`, `output_restart_coordinates()`, and `output_final_coordinates()`. Which of these functions to call is based on the parameters contained in the `SimParameters` object and the current timestep.
- `void velocity(int timestep, int n, Vector *vel)`
  This function takes the velocities for timestep `timestep` from the array `vel` and calls the appropriate output routines. `n` is used to specify the number of velocities that are passed in. At the moment, the output routines called include `output_restart_velocities()`, and `output_final_velocities()`. Which of these functions to call is based on the parameters contained in the `SimParameters` object and the current timestep.
- `void long_force(int timestep, int n, Vector *force)`
  Produce a long range electrostatic force DCD file.
- `void short_force(int timestep, int n, Vector *force)`
  Produce a short range electrostatic force DCD file.
- `void all_force(int timestep, int n, Vector *force)`
  Produce DCD file recording the total force on each atom.
- `void initialize_vmd_connection()`
  Starts up the VMD connection.
- `void close_vmd_connection()`
  Closes the VMD connection.
- `void vmd_process_events()`
  Process events from VMD.
- `void send_vmd_static(mdc_app_arena *arena)`
  Sends static data to VMD.
- `void send_vmd_dyn(mdc_app_arena *arena)`
  Sends dynamic data to VMD.

- `void print_vmd_static_data()`
  Debugging routine to print VMD static information to the screen.

- `void recv_vmd_patch_loads()`
  Collects Patch load messages.

5.3.27 Pairlist

Purpose:
This object is used to store the electrostatic pairlist information. It implements a storage scheme using a two dimensional array. The main storage consists of a set of equally sized segments. Each segment is an array of pairlist values. The array of segment pointers is used to keep track of these segments. If more storage is needed, additional segments are allocated. The reason for such a scheme is to reduce the number of memory allocations and deallocations, i.e., instead of for each pair, memory allocation is done for a number of them (a segment).

Derived From:
Nothing.

Files:
`Pairlist.h`

Constructor:
`Pairlist(int numPairs)`: Initializes `Pairlist` object. `numPairs` is an estimate for maximum number of entries that could be stored in this pairlist. This number is used to calculate the segment size. A bad estimate will not cause any failure but it will result in too small or too large segments. Currently, the segment size is calculated as `numPairs`/20 .

Destructor:
~`Pairlist(void)` : Frees the segments and the segment array.

Method of Use:

There is a small set of public routines provided. After creating a `Pairlist` object, entries can be added by invoking `add()` function. After all the pairs are entered, the `Pairlist` object can be traversed in a sequential order by invoking functions `head()` and `next()`. `head()` resets the traversal position to the beginning of the list and returns the first entry. Subsequent entries are returned by the `next()` function. The `reset()` function initializes the pairlist without freeing already allocated segments for reuse (due to performance reasons).

Defined Constants:
`MIN_SEG_SIZE` segment size can't go below this value
`ARRAY_INC_SIZE` size of chunks to add to array of segment pointers

Functions:

- `int size()`
  returns the number of entries in the list.

- `void reset()`
  resets the pairlist without freeing the memory for reuse.

- `void add(int a1, int a2, BigReal kqq, BigReal vA, BigReal vB, char oc)`
  adds a pair (with the values provided) to the pairlist.
- `const PLValues *head()`
  resets the traversal position to the beginning and returns a pointer to the first pair. If the pairslist is emtpy, it returns `NULL`
- `const PLValues *next()`
  advances to the next pair. it returns a pointer to the pair or `NULL` if there is no more pair.

5.3.28 Parameters

Purpose:
The `Parameters` class is used to read in, store, and find parameters from X-PLOR style parameter files. Parameters read in include linear bond parameters, angle bond parameters, dihedral bond parameters, improper bond parameters, single atom Van der Waals parameters, and pairwise Van der Waals parameters. Multiple parameter files may be read in for a given simulation. All of the parameters are stored in internal data structures that are efficient for later for these parameters both during the reading of the molecular structure and during the simulation.

A `Parameters` object exists in each process. The master node will read its information directly from the parameter file(s) and distribute this information to all the other processes. This object will then be used to find parameters as they are needed during the simulation.

Derived From:
Nothing.

Files:
`Parameters.h, Parameters.C, structures.h`

Constructor:
`Parameters()`

Destructor:
`~Parameters(void)`

Method of Use:
For speed, it is desirable to be able to access the parameters in constant time during the simulation, implying that the parameters should be stored in arrays and accessed via an index. But because multiple parameter files with an unknown number of parameters must be read in and these parameters must be searched by atom type during the reading of the .psf file, it would be nice to have the data in a dynamic data structure that can be searched quickly by atom type. To accomodate both of these goals, the data is first read into dynamic data structures. These data structures are converted to arrays once all the parameter files are read and these arrays are used for constant time access to parameters during the simulation. When the dynamic data structures are queried during the reading of the .psf file, the index into the arrays of parameters is returned. Once the .psf file has been read in, the dynamic data structures are then destroyed and only the arrays of parameters are used.

In order to accomodate the different data structures used, the reading of the parameter files must occur in several stages. First, all of the parameter files are read in using the function `read_parameter_file(char *name)`. The parameters will be stored in dynamic data structures that can quickly be searched by atom type. Once all of the parameter files have been read in, the function `done_reading_files()` is used to copy all of the parameters from these dynamic data structures to static arrays and an index into these arrays are assign to each parameter. Then the .psf file is read in. For each atom or bond, the `Parameter` object queries the dynamic data structure by atom type, and returns an index into the new static arrays using the `assign_*()` functions. Once the .psf file has been read, the function `done_reading_structure()` is called. This will destroy the dynamic data structures. From this point on, only the static arrays of parameters an the indexes into these arrays will be used. The `get_*()` functiones are used to access these arrays by index.

The `send_Parameters()` and `receive_Parameters()` functions are used to send the parameters from the master node to the client nodes and to receive the parameters on the client nodes.

Defined Types:

- `Index` – This type is used to index into the arrays of parameter information. Right now, it is typedef'd to an unsigned short, which allows for up to 32767 different parameters, which should be plenty. If it ever becomes necessary, this can be defined to a larger type if more parameter types are needed.

Functions:

- `void read_parameter_file(char *name)`
  Read in all of the parameters in the file specified by `name` and store them for later use. This routine can be called multiple times with different file names if there are multiple parameter files to be read.
- `void done_reading_files()`
  Signals the parameter object that all of the parameter files have been read in.
- `void done_reading_structure()`
  Signals the parameter object that the structure file has been read in.
- `void assign_vdw_index(char *type, Atom *atom)`
  Find the `Index` for the Van der Waals parameters for the atom type specified by `type` and assign it to the appropriate field in the structure specified by `atom`.
- `void assign_bond_index(char *atom1, char *atom2, Bond *bond)`
  Find the `Index` for the bond parameters for a bond between atom types `atom1` and `atom2` and assign this `Index` to the appropriate field in the structure specified by `bond`.
- `void assign_angle_index(char *atom1, char *atom2, char *atom3, Angle *angle)`
  Find the `Index` for the angle parameters for an angle bond between atoms types `atom1`, `atom2`, and `atom3` and assign this `Index` to the appropriate field in the structure specified by `angle`.
- `void assign_dihedral_index(char *atom1, char *atom2, char *atom3, char *atom4, Dihedral *dihedral)`
  Find the `Index` for the dihedral parameters for a dihedral bond between atoms types `atom1`, `atom2`, `atom3` and `atom4` and assign this `Index` to the appropriate field in the structure `dihedral`.
- `void assign_improper_index(char *atom1, char *atom2, char *atom3, char *atom4, Improper *improper)`
  Find the `Index` for the improper parameters for a improper bond between atoms types `atom1`,

atom2, atom3 and atom4 and assign this Index to the appropriate field in the structure specified by improper.

- void send_Parameters(Communicate *com_obj)
  Send the parameters from the master node to the client nodes using the communicate object com_obj.

- void receive_Parameters(Message *msg)
  Read the parameters on the client nodes from the message sent by the master node.

- void get_bond_params(Real *k, Real *x0, Index index)
  Retreive the parameters k and x0 for the bond with index index.

- void get_angle_params(Real *k, Real *theta0, Index index)
  Retreive the parameters k and theta for the angle bond with index index.

- int get_improper_multiplicity(Index index)
  Retreive the multiplicity for the improper bond with index index.

- int get_dihedral_multiplicity(Index index)
  Retreive the multiplicity for the dihedral bond with index index.

- void get_improper_params(Real *k, int *n, Real *delta, Index index)
  Retreive the parameters k, n, and delta for the improper bond with index index.

- void get_dihedral_params(Real *k, int *n, Real *delta, Index index)
  Retreive the parameters k, n, and delta for the dihedral bond with index index.

- void get_vdw_params(Real *sigma, Real *epsilon, Real *sigma14, Real *epsilon14, Index index)
  Retreive the parameters simga, epsilon, sigma14 and epsilon14 for the atom with index index.

- int get_vdw_pair_parameters(Index atom1, Index atom2, Real *sigma, Real *epsilon, Real *sigma14, Real *epsilon14)
  Find a the pair specific Van der Waals parameters for atom types atom1 and atom2. This is the only find routine that will NOT terminate the program if a match is not found. Instead, a 1 is returned if a match is found, or a 0 otherwise. If a match is found, the values specified by sigma, epsilon, sigma14, and epsilon14 are populated with the appropriate parameters.

- void print_bond_params()
  Print out a list of all the bond parameters known to this object.

- void print_angle_params()
  Print out a list of all the angle parameters known to this object.

- void print_dihedral_params()
  Print out a list of all the dihedral parameters known to this object.

- void print_improper_params()
  Print out a list of all the improper parameters known to this object.

- `void print_vdw_params()`
  Print out a list of all the Van der Waals parameters known to this object.
- `void print_vdw_pair_params()`
  Print out a list of all the Van der Waals pair specific parameters known to this object.
- `void print_param_summary()`
  Print out a summary of the number of each type of parameter that has been read in.

5.3.29 ParseOptions

Purpose:
This class serves as a way to parse a ConfigList. It is designed to simplify the SimParameters class by providing a uniform mechanism to determine which options are valid at which times. It is rather limited as future NAMD development will tend towards run-time communications with a more general controller; so a general parser is not required. This class is used in four phases; set up the dependencies, verify the internal consistancy, read in the ConfigList, get or use the defined variables.

The first stage tells the class which options are dependent on others. Only one dependency is allowed. The dependent option is called the "child" and the option it is dependent upon is the "parent". The dependencies can be expressed as a tree with the option "main" defined as the uppermost parent. (Hence, all other options are dependent on main.) In addition, it can be used to define a range for which a given option is valid as well as give it units.

The second checks that all elements are derived from the "main" and that there are no loops.

The third reads the elements from the ConfigList and sets the appropriate options. If desired, a pointer can be passed into ParseOptions which will also be set at this time. If options are out of range or too many are given, the error is printed. If extra or unknown options are defined in the ConfigList, they are printed as warnings.

The fourth is used to read options which were set. This is in some sense optional as the same information can be extracted through the variable pointer passed in the first stage.

Derived From:
Nothing.

Files:
`ParseOptions.C, ParseOptions.h`

Constructor:
`ParseOptions(void)`

Destructor:
`~ParseOptions(void)`

Enumerations:

- Range: FREE_RANGE, POSITIVE, NOT_NEGATIVE, NEGATIVE, NOT_POSITIVE
- Units: UNIT, FSEC, NSEC, SEC, MIN, HOUR, ANGSTROM, NANOMETER, METER, KCAL, KJOULE, EV, KELVIN, UNITS_UNDEFINED

Global Functions:

- const char *string(Range r) : returns a name for the given range

- const char *string(Units u) : returns a name for the given unit
- BigReal convert(Units to, Units from): returns the scaling factor needed to convert from 'from' to 'to'

Defines:

- PARSE_BIGREAL : used to define a FLOAT without giving a pointer (same as "(BigReal *) NULL")
- PARSE_FLOAT : the same as PARSE_BIGREAL
- PARSE_VECTOR : similarly defines a VECTOR (as "(Vector *) NULL")
- PARSE_INT : similarly defines an INT
- PARSE_UINT : similarly defines an UINT (unsigned int)
- PARSE_BOOL : similarly defines an INT (which is boolean)
- PARSE_STRING : defines a STRING (as "(char *) NULL")
- PARSE_ANYTHING : defines a STRINGLIST
- PARSE_MULTIPLES : defines a STRINGLIST that can have multiple elements (same as "(StringList **) NULL, TRUE")

Method of Use:
This parser can be used for data which is "dependent" on other data. For instance, the option "FMALevels" is dependent on the option "FMAOn"; if the latter (the parent) is not given in the input file, then the former (the child) is never needed. There are two levels of dependency; "require" and "optional." In the given example, if it is required that FMALevels be given in the configuraion file whenever FMAOn is true, then the appropriate code would be:

```
ParseOptions opts;
...
opts.require("FMAOn", "FMALevels", "Number of levels in the FMA exansion");
```

The first term is a char * with the name of the parent, the second term is the name of the new option. Names are not case sensitive. The third field is the message displayed when a warning or error occurs which involves the newly defined option. The special parent name "main" is used for options which are not dependent on other options. Now, if FMALevels is optional, the code would look like:

```
opts.optional("FMAOn", "FMALevels", "Number of levels in the FMA exansion");
```

These functions are overloaded and can take several more parameters. The first passes a pointer (int *, BigReal *, Vector *, char *, or StringList **) to the ParseOptions class. Then, when the values are set from the ConfigList, this pointer is used to set the appropriate variable automatically. For the int, BigReal, and Vector types, the next term (if it exists) defines a default value which is used if the option is not defined in the configuration file but the parent is defined. There is no way to give a default value for char * or StringList. The “set” for a char * is actually a strcpy, so you need to have allocated space.

Using the same example, suppose the value of “FMALevel” is to be placed in the variable “num_fma_levels.” If there is no default value and the term is required, then the code is:

```
int num_fma_levels;
opts.require(“FMAOn”, FMALevels”, “Number of levels in the FMA expansion”,
&num_fma_levels);
```

If instead the default number is 5, the code would be:

```
opts.require(“FMAOn”, FMALevels”, “Number of levels in the FMA expansion”,
&num_fma_levels, 5);
```

Functionally there is no difference between “require” and “optional” if a default is given. Also, if the pointer is NULL, nothing bad happens. This is useful if you want an option to have a range but don’t want to use it immediately.

There is a special form of these functions for boolean values. NAMD does not have a real “boolean” variable; it has “typedef Bool int”, which is indistinguishable from int to the compiler. Since it much nicer to be able to say “FFTOn yes” than “FFTOn 1” the functions ’requireB’ and ’optionalB’ were made. Boolean terms do more than define a “yes/no” value (which, by the way, is internally handled as an integer); they are also used to turn on other blocks of code.

Now normally, if the option is defined in the configuration file and other data is dependent on that parent, then those dependencies are checked. Using that definition, setting “FMAOn” to either “on” or “off” will tell the ParseOptions to check terms like “FMALevels”, which are dependent on that parent. To get around that problems, if a boolean value is a parent and is defined false, then it is undefined. This makes “FMAOn no” identical to not listing FMAOn at all.

Here is how the dependencies could look for FMA:

```
int FMAOn, FMAlevels, FMAMp, FMAFFTOn, FMAFFTBlock;
opts.optionalB("main", "FMA", "Should FMA be used?", &FMAOn);
opts.require("FMA", "FMALevels", "Tree levels to use in FMA", &FMAlevels, 5);
opts.require("FMA", "FMAMp", "Number of FMA multipoles", &FMAMp, 4);
opts.requireB("FMA", "FMAFFT", "Use FFT enhancement in FMA?", &FMAFFTOn,
FALSE);
opts.require("FMAFFT", "FMAFFTBlock", "FFT blocking factor", &FMAFFTBlock,
4);
```

By default, at most one option of a given name is allowed. If there are multiple definitions, ParseOptions prints that fact to namdErr. Very few options allow multiple inputs of the same option; the only one at this time is “parameters”. The only way to define this is through the

StringList version of optional/ require. The first three terms are identical to the other, similar functions, and the fourth takes a StringList **. The fifth term, which by default is FALSE, defines if multiple instances are allowed. For example, the following allows multiple "parameters":

```
opts.require("main", "parameters", "one entry for each CHARMm 19 or CHARMm 22
compatable force field file", (StringList **) NULL, TRUE);
```

Many types of input must be positive, for instance, timestep size and number of steps per cycle. This range checking can be done in the ParseOptions class with the "range" function. This takes the option name and the range it can take (the valid ranges are listed in the enumerations section). Here's an example of how to define that FMALevels must be positive:

```
opts.range("FMALevels", POSITIVE)
```

Some of the options represent physical values, which have units. The options can assume that the input will always have a specified unit, and the user must use those units by default. However, if the parser understands some common units (like "fs" and "nm") then the user can specify those directly. The funtions named 'units' get and set the units associated with the given option. During the parsing, the user input gets translated as need be so that it is in the specified units. Here's how the code would look:

```
opts.units("timestep", FSEC);   // timesteps are in femtoseconds
```

Then the option "timestep 1" returns a value of 1, as do "timestep 1 fs" and "timestep 0.001ns."

After the dependency, range, and units functions are defined, the ParseOptions should be checked to ensure there are not cyclic dependencies and that all options are accesible from "main". This is done with the check_consistancy function, which returns TRUE if the system contained no errors.

Once that is done, the values can be set via the function "set", which is passed the ConfigList. This does the dependency and range checks as well as set any variables which may have been passed in via a pointer. If there was a range error, it prints (to namdErr) the reason for the error as well as the message associated with that option. Errors related to units conversion are also printed at this time. If there was an option in the ConfigList that was not in the language, it prints (to namdWarn) all such unknown options. It also prints (to namdWarn) all options which the keyword is known, but not needed (as when FMAOn is off but FMALevels is defined). The last several functions would look like:

```
if (!opts.check_consistancy()) {
namdErr << "Internal parsing unsuccessful" << sendmsg;
return 1;
}
ConfigList clist("test.namd"); // open and read the file
if (!clist.okay()) {
namdErr << "Cannot read 'test.namd'" << sendmsg;
return 1;
}
if (!opts.set(clist)) { // feed it through the input parser
namdErr << "There were errors in the configuration file" << sendmsg;
return 1;
}
```

There are several ways to access the information. As mentioned earlier, it is possible to have variables set automatically via pointers passed during the dependency definition. It is also possible to use one of the "get" functions. These all take as the first parameter the name of the option and, for the second parameter, a pointer of where to put the information. ParseOptions knows the data type of that option from the dependency definition and can perform type conversion for most cases; the exceptions are listed in the definition section. All these functions return 1 if the function was successful and prints (to namdWarn) a warning if type conversion took place.

There are two ways to access data with multiple values; as a StringList * or through a char *. "get(char *s, int n=0)" takes as an optional parameter the index of the string to return in the StringList. The total number of elements in a StringList is accessible via "num".

For example, the following code could be used to list the given parameter files:

```
int num = opts.num("parameters");
char s[100];
for (int i=0; i<num; i++) {
opts.get("parameters", s, i);
namdInfo << " " << s << sendmsg;
}
```

Finally, "defined" tells if a given option was defined during the "set" and "exists" tells if a given option was stated in the dependencies.

Private Functions:

- `void add_element(DataElement *)`
  Adds a new DataElement to the internal array (see Internal Classes, below)
- `int make_dependencies(DataElement *)`
  Find if the new DataElement is dependent on, or depends on, any of the DataElements already in the array.
- `atoBool`
  Convert a string to Bool. Returns 1 if yes, on, or true; 0 if no, off, or false, and -1 if anything else
- `Boold is_parent_node(DataElement *)`
  Returns TRUE if the given DataElement has any children. Used to determine if a BOOLEAN which is FALSE should be undefined.

Public Functions:

- `int require(const char *newname, const char *parent, const char *msg, BigReal *ptr, BigReal default)`
  A required BigReal, with a default

- `int require(const char *newname, const char *parent, const char *msg, BigReal *ptr)`
  A required BigReal with no default
- `int require(const char *newname, const char *parent, const char *msg, Vector *ptr, Vector default)`
  A required Vector, with a default
- `int require(const char *newname, const char *parent, const char *msg, Vector *ptr)`
  A required Vector with no default
- `int require(const char *newname, const char *parent, const char *msg, int *ptr, int default)`
  A required int with a default
- `int require(const char *newname, const char *parent, const char *msg, int *ptr)`
  A required int with no default
- `int requireB(const char *newname, const char *parent, const char *msg, int *ptr, int default)`
  A Boolean (variant of int) with a default
- `int requireB(const char *newname, const char *parent, const char *msg, int *ptr)`
  A Boolean (variant of int) with no default
- `int require(const char *newname, const char *parent, const char *msg, unsigned int *ptr, unsigned int default)`
  A required unsigned int with a default
- `int require(const char *newname, const char *parent, const char *msg, unsigned int *ptr)`
  A required unsigned int with no default
- `int require(const char *newname, const char *parent, const char *msg, StringList **ptr = NULL, int many_allowed = FALSE)`
  A required option with a StringList return. If TRUE, the second parameter allows StringLists with more than one element. There are no defaults for StringLists
- `int require(const char *newname, const char *parent, const char *msg, char *ptr)`
  A required option with a char * return. There are no defaults with a char *. If the option exists in the ConfigList, it is copied (via strcpy) to ptr.
- `int optional(const char *newname, const char *parent, const char *msg, BigReal *ptr, BigReal default)`
- `int optional(const char *newname, const char *parent, const char *msg, BigReal *ptr)`

- `int optional(const char *newname, const char *parent, const char *msg, Vector *ptr, Vector default)`
- `int optional(const char *newname, const char *parent, const char *msg, Vector *ptr)`
- `int optional(const char *newname, const char *parent, const char *msg, int *ptr, int default)`
- `int optional(const char *newname, const char *parent, const char *msg, int *ptr)`
- `int optionalB(const char *newname, const char *parent, const char *msg, int *ptr, int default)`
- `int optionalB(const char *newname, const char *parent, const char *msg, int *ptr)`
- `int optional(const char *newname, const char *parent, const char *msg, unsigned int *ptr, unsigned int default)`
- `int optional(const char *newname, const char *parent, const char *msg, unsigned int *ptr)`
- `int optional(const char *newname, const char *parent, const char *msg, StringList **ptr = NULL, int many_allowed = FALSE)`
- `int optional(const char *newname, const char *parent, const char *msg, char *ptr)`
- `void range(const char *name, Range newrange)`
  Define that the variable associated with name has the given range
- `Range range(const char *name)`
  Return the range associated with the given name
- `Bool units(const char *name, Units units)`
  Associate units to the given option. Prints a warning if this isn't a FLOAT option
- `Bool units(const char *name, Units *units)`
  Return the units associated with the given name
- `Bool scan_float(DataElement *el, const char *s)`
  Read in a BigReal. If there are units, do the necessary conversion. Put the final result in el→fdata. Returns error value
- `Bool scan_vector(DataElement *el, const char *s)`
  Read in a Vector and return error value Put the final result in el→fdata. Returns error value
- `Bool scan_int(DataElement *el, const char *s)`
  Read in an int and put the result in el→idata. Returns error value

- `Bool scan_uint(DataElement *el, const char *s)`
  Read in an unsigned int and put the result in el→uidata. Returns error value

- `Bool scan_bool(DataElement *el, const char *s)`
  See if the string can be parsed as TRUE or FALSE and sets el→idata to TRUE or FALSE, respectively. Returns error value

- `Bool set_float(DataElement *)`
  Make sure the BigReal data value is in range (Returns FALSE if not) and sets the BigReal *, if appropriate.

- `Bool set_vector(DataElement *)`
  If the Vector pointer is defined, set it to the current value

- `Bool set_int(DataElement *)`
  Same as set_float, but for an integer

- `Bool set_uint(DataElement *)`
  Same as set_float, but for an unsigned integer

- `void set_bool(DataElement *)`
  Sets the int data value and sets the int *, if appropriate

- `void set_stringlist(DataElement *)`
  Sets the StringList **, if appropriate

- `void set_string(DataElement *)`
  Copies the first element of the given ConfigList to the sptr, if appropriate

- `Bool check_consistanct(void)`
  Check that there are no internal loops or other errors in the DataElement array.

- `Bool set(const ConfigList& configlist)`
  Reads each element of the configlist, checks the internal DataElement array, sets the appropriate data pointers, prints all warnings and/or errors, and returns TRUE if it all worked out.

- `DataElement *internal_find(const char *name)`
  Find the element in the internal array corresponding to the given name, or NULL if it doesn't exist.

- `Bool get(const char *name, int *val)`
  Sets the pointer based the integer value associated with the given name, doing conversion if necessary. If conversion was needed, prints the warning (or error, if it is a Vector) to the screen. Returns FALSE if the name doesn't exist

- `Bool get(const char *name, BigReal *val)`
  Same, as get(..., int), but for BigReal

- `Bool get(const char *name, Vector *val)`
  Sets the pointer based on the Vector value associated with the name. It can only do conversions from STRING and STRINGLIST. Returns FALSE if the conversion is not possible.

- `Bool get(const char *name, StringList **val)`
  Sets *val to configList→find(name), if it exists, or returns FALSE.

- `Bool get(const char *name, char *val, int n=0)`
  Copies the nth element of the configList to val, or returns FALSE.

- `Bool exists(const char *name)`
  Returns TRUE if an element with the name 'name' exists in the internal DataElement array

- `Bool defined(const char *name)`
  Returns TRUE if 'name' exists in the array and was given either a default or was in the ConfigList

Internal Class:
`ParseOptions::DataElement` – DataElement is little more than glorified typedef. It is completely public to ParseOptions and is private to the rest of the code. Its only functions are constructors, which are used to set the correct values and ensure that other values are sane.

This class stores all the information needed to describe the data element needed by ParseOptions. It know the name of the datum, its parent's name, the pointer to the parent (if it exists), whether the datum is optional or required, the type of the data (through run-time data typing), the default value, if it exists, and so on. Inside ParseOptions, the DataElements are stored as an array which grows (during add_element) as needed.

Derived From:
Nothing.

Enumeration:
data_types: UNDEF, FLOAT, VECTOR, INT, BOOL, STRINGLIST, STRING - the type used for the run-time typing

Constructor:

- DataElement(const char *newname, const char *newparent, int optional, const char *err, BigReal *ptr, BigReal default)
  Make a FLOAT type with the given default value which can set ptr

- DataElement(const char *newname, const char *newparent, int optional, const char *err, BigReal *ptr)
  Ditto, but with no default value

- DataElement(const char *newname, const char *newparent, int optional, const char *err, Vector *ptr, Vector default)
  Make a VECTOR type with the given default value which can set ptr

- DataElement(const char *newname, const char *newparent, int optional, const char *err, Vector *ptr)
  Ditto, but with no default value

- DataElement(const char *newname, const char *newparent, int optional, const char *err, int *ptr, int default)
Make an INT type with default value and can set ptr

- DataElement(const char *newname, const char *newparent, int optional, const char *err, int *ptr)
Ditto, with no default value

- DataElement(const char *newname, const char *newparent, int optional, const char *err, unsigned int *ptr, unsigned int default)
Make an UINT type with default value and can set ptr

- DataElement(const char *newname, const char *newparent, int optional, const char *err, unsigned int *ptr)
Ditto, with no default value

- DataElement(const char *newname, const char *newparent, int optional, const char *err, StringList **ptr, int many_allowed = FALSE)
Make a STRINGLIST type which may or may not have multiple elements

- DataElement(const char *newname, const char *newparent, int optional, const char *err, char *ptr)
Make a STRING type which can strcpy to ptr

5.3.30 Patch

Purpose:
Each `Patch` object is responsible for maintaining a region of space in the simulation. This includes maintaining the current positions and velocities of all the atoms in this region, calculating all the forces acting on these atoms, and integrating the equations of motion for these atoms during each timestep to update their positions and velocities.

Derived From:
Nothing.

Files:
`Patch.h, Patch.C`

Constructor:
There are two constructors for this class. This first is used when a `Patch` object is created from scratch. It creates an empty patch with no atoms. The second constructor is used when the `Patch` object being created is a patch that has migrated from another processor. Currently, only the first one is implemented since patch migration has not yet been implemented.

- `Patch(int, PatchList *)`: New empty patch, with no atoms. It is passed its patch id and a pointer to the `PatchList` object that owns it.
- `Patch(int, PatchList *, Message *)`: Constructs a patch from data in the given Message. After construction, deletes the Message. The format for this Message is given below.

Destructor:
`~PatchList(void)`

Method of Use:
The `Patch` object is responsible for managing most of the work done during the simulation. It owns a region of space and is responsible for all of the atoms in that region of space. This responsibility includes maintaining the current positions and velocities for these forces, calculating or gathering all of the forces necessary to do intrgration during each timestep, and having the ability to send out and receive atoms that have moved between regions of space at the end of a cycle. It is assumed that hydrogen atoms are contained in the same patch as the heavy atoms to which they are attached. Atom reassignment occurs in such a way as to guarantee that that will remain the case.

To accomplish these things, each `Patch` object contains a number of other objects to manage these operations. These objects include force objects that are repsonsible for calculating various components of the force field for the local atoms and an `Integrate` object that is responsible for integrating the forces of motion and updating the current positions and velocities of each atom during each timestep.

The `Patch` object's main purpose is to provide a control and data framework to allow each of these objects to do their work and allow the simulation to continue. To this end, each timestep follows a basic sequence of steps. These steps are:

1. **Send out coordinate messages -** This is accomplished using the function `send_msgs()` and it involves sending out local coordinates to neighboring patches so that interactions with neighbors can be calculated. This also involves sending a message to ourself telling the patch to perform local force calculations. During normal, mid-cycle steps, this step actually occurs at the end of the previous timestep rather than at the begining of a new timestep. For more details on the communication pattern, see section 3.1.

2. **Process incoming messages -** This includes receiving and processing messages that contain all coordinates, only bonded coordinates, or messages that contain forces calculated by another patch, as well as handling messages that are sent to itself to trigger local force calculations and integration. During this stage, the number of outstanding messages is also monitored. When all excpected messages have been processed, a message is sent to ourself telling us to perform integration.

3. **Integration and energy reporting -** This is really just a specialized case of the previous case, but it represents the last calculation performed during a timestep. At this point, all of the forces for the current timestep have been evaluated, and the equations of motion are integrated by the `Integrate` object to obtain new values for the position and velocity of all the local atoms. Then the energy from all the force objects are gathered and the kinetic energy is calculated and these values are passed to the local `Collect` object so that they can be summed across all processors.

The rest of the functionality in the `Patch` object is to deal with adminstrative things such as startup and shutdown, the transfer of atoms from patch to patch, recalibration of force objects at the cycle boundaries, etc.

Private Data Types:
These data structures are used only within the `Patch` object.

- `BondedWithNeighbor` - This data structure is used to track the atoms that need to be sent to each neighbor that we only send bonded coordinates to. Sending coordinates to these neighbors is basically a gathering operation where a specific number and sequence of atom coordinates are gathered and sent during each timestep. `BondedWithNeighbor` contains:
  - `num` - The number of atoms that are to be sent to this neighbor.
  - `atoms` - An array of size `num` that contains the local atoms indexes of the atoms that are to be sent to this neighbor.
  - `coords` - An array of size `num` of Vectors that is used to gather the coordinates to be sent to this neighbor. This provides a buffer to gather the positions in during each timestep so that space doesn't need to be allocated and freed during each timestep.

So the act of gathering the atoms for a neighbor given a `BondedWithNeighbor` structure can be accomplished by using the following loop:

```
BondedWithNeighbor bwn;
int i;

for (i=0; i<bwn.num; i++)
{
   bwn.coords[i] = x[bwn.atoms[i]];
}
```

- `ReturnForceVectors` - This structure is used to hold force vectors to be returned to those neighbors that send us all their atom coordinates. The forces to be returned are accumulated here, and then sent back to that neighbor. `ReturnForceVectors` contains:
  - `num` - The number of force vectors to return to this neighbor.
  - `forces` - An array of size `num` of Vectors to accumulate the forces to be returned to the neighbor in. This array is set to contain all 0's at the begining of each timestep. When the coordinate message from this neighbor is processed, these forces are populated and then sent back to the neighbor.
- `AtomMsgList` - This structure is used to form a linked list of atom reassignment `Messages` that have been received. During atom reassignment, all the atom reassignment messages received from neighbors that actually contain new atoms for this patch are buffered in a list using `AtomMsgList` structures. Once all the messages have been received, all the messages are processed at once. `AtomMsgList` contains:
  - `num` - Number of atom contained in this message.
  - `msg` - A pointer to the `Message` object itself.
  - `next` - A pointer to the next link in the list.

Private Data:
The `Patch` object is where all the important data for actually doing the calculations for the simulation resides. This section details all of the data contained within a `Patch` object.

- `PatchList *parentList` - Pointer to the `PatchList` object that owns this patch. This is used to access items contained in the `PatchList` object easily.
- `int myId` - Patch number for this patch. This number is set during construction, and never changes during the life of the patch.
- `int numAtoms` - Number of atoms residing on this patch.
- `int currentTimestep` - The timestep that this `Patch` is currently working on. Currently, patches that reside on the same node will always being working on the same timestep. Patches that reside on different nodes may be working on different timesteps. Thus, at any time, any neighboring patch may be working on a different timestep than this patch.

- `int *atoms` - Array of size `numAtoms` that contains the *global* indexes for the local indexes. Thus `atoms[i]` contains the global atom index for the ith local atom.
- `Vector *x` - Array of vectors containing the current positions of all the local atoms.
- `Vector *v` - Array of vectors containing the current velocities of all the local atoms.
- `Vector *f` - Array of vectors containing the current forces acting on the local atoms. This array is set to all 0's at the begining of each timestep.
- `Vector *xOrig` - Positions at start of timestep. This is used for pairlist violation checking.
- `BigReal safetyMargin2` - $[\frac{1}{2}(\texttt{pairlistdist -cutoff})]^2$, used for determination of a pairlist margin violation.
- `BigReal excludedEnergy` - Electrostatic energy from exclusions.
- `Vector *localAndExclElec` - Electrostatic forces from local interactions and exclusions.
- `int numSend` - Number of neighboring patches that we send all coordinates to.
- `int numRecv` - Number of neighboring patches that we receive all coordinates from.
- `int *sendNeighbors` - Array of length `numSend` that contains the patch ids of the neighbors that we send all coordinates to.
- `int *recvNeighbors` - Array of length `numRecv` that contains the patch ids of the neighbors that we received all coordinates from.
- `BondedWithNeighbor *bondedInfo` - Array of length `numRecv` that contains the information about the bonded coordinates that we need to send to each neighbor that we send only bonded coordinates too. The information is placed into this array in the same order as the patch ids are placed in `recvNeighbors`. In other words, if patch id `p` satisfies `p=recvNeighbors[i]`, then the information about the bonded coordinates to send to patch `p` is contained in the structure `bondedInfo[i]`.
- `ReturnForceVectors *returnForces` - Array of length `numSend` that provides a place to accumulated forces that are to be returned to a neighboring patch that sends this patch all of its atoms. As with `bondedInfo`, the entries in `returnForces` correspond to the patch id of the corresponding element in `sendNeighbors`.
- `Bool doneLocal` - Flag which indicates whether the local calculations have been performed during the current timestep yet or not.
- `Bool sentPreIntegrate` - Flag which indicates whether the preintegrate message has been sent.
- `int coorMsgsOutstanding` - Counter which is used to determine how many coordinate messages we are still waiting for during the current timestep. At the begining of each timestep, this value is set to `numSend+numRecv`. As each bonded coordinate and all coordinate message is processed, this value is decremented by one. When this value reaches 0, the patch has processed all of the coordinate messages for the current timestep.

- `int forceMsgsOutstanding` - Counter similar to `coorMsgsOutstanding` that is used to track the number of force messages that the patch is waiting for. At the begining of each timestep, it is set to `numSend`. When its value reaches 0, all of the force messages for the current timestep have been processed.
- `int atomMsgsOutstanding` - Counter that is used to track the number of atom reassigment messages we are waiting for. At the begining of atom reassignment, this value is set to the total number of neighboring patches. When it reaches 0, all of the expected messages have been received.
- `int numAtomsRemoved` - This value is only valid during atom reassignment, and it contains the number of atoms that have left the spatial region owned by this patch during the current reassignment period.
- `int numAtomsAdded` - This value is also only valid during atom reassignment, and it contains the number of atoms that have entered the spatial region owned by this patch during the current atom reassignment period.
- `LintList *atomsRemoved` - `LintList` object that is created for each atom reassignment that stores the local indexes of atoms that have left the region of space owned by this patch.
- `AtomMsgList *atomsReceived` - Linked list that contains messages from neighbors who have sent us atoms that have migrated into the region of space owned by this `Patch`. This list is created and destroyed during each atom reassignment.
- `AtomMsgList *atomListTail` - Pointer to the tail of `atomsReceived` so that new messages can be added to the list in constant time.
- `int tsType` - Flag that has the value of either `BEGIN_CYCLE` or `MID_CYCLE` which indicates whether this is the first timestep in a cycle or not. This is important, since during the first timestep of each cycle, the messages received are slightly different and the actions performed on each message is slightly different as initialization is done by each force object. The `MID_CYCLE` steps all perform the same operations.
- `constraintsActive` - Flag which indicates if constraints are on.
- `sphericalBCActive` - Flag which indicates if spherical boundary conditions are on.
- `eFieldActive` - Flag which indicates if the constant applied electric force is one.
- `fmaActive` - Flag which indicates if FMA is being used.
- `fullDirectActive` - Flag which indicates if full direct electrostatic evaluation is being used.
- `atomsChanged` - Flag which indicates if atoms on this patch changed during the last reassignment.
- `BondForce *bondForce` - Force object responsible for calculating linear bond interactions.
- `AngleForce *angleForce` - Force object responsible for calculating angular bond interactions.

- `ElectForce *electForce` - Force object responsible for calculating short range electrostatic and Van der Waals interactions.
- `FieldForce *fieldForce` - Force object responsible for calculating the applied electric field forces.
- `ImproperForce *improperForce` - Force object responsible for calculating the improper bond interactions.
- `DihedralForce *dihedralForce` - Force object responsible for calculating the dihedral bond interactions.
- `ConstraintForce *consForce` - Force object responsible for calculating the forces on constrained atoms.
- `SphericalBCForce *sphereForce` - Force object responsible for calculating the spherical boundary condition forces.
- `LongForce *longForce` - Force object responsible for calculating the long range electrostatic interactions.
- `Integrate *integrator` - Integration object responsible for integrating the equations of motion to produce new positions and velocities.

Public Functions:
Each `Patch` object is owned and managed by a `PatchList` object. Thus, almost all calls to these public functions are done by the `PatchList` object.

- `void get_initial_positions(Message *msg)`
  Process a message containing the initial positions of atoms belonging to this patch. This is only used at the very begining of the simulation as atoms are passed out to patches.
- `void get_initial_velocities(Message *msg)`
  Process a message containing the initial velocities of atoms belonging to this patch. This is only used at the very begining of the simulation as atoms are passed out to patches.
- `void send_msgs(int timestep)`
  Send out necessary coordinate and self calculation messages to neighboring patches and ourself.
- `void process_msg(Message *msg, int tag)`
  Process the message `msg` that was received with the tag `tag`. It is in processing these messages that most of the work of the patch is performed.
- `void copy_self(Message *msg)`
  Place a copy of all the useful data in this patch into the message `msg` so that this patch can migrate to another processor.
- `void initialize_timestep(int timestep)`
  Prepare the patch to perform a new timestep.

- `void end_cycle()`
  Tear things down at the end of a cycle. This includes things like freeing all the force objects, reseting counters, etc.

- `int getNumAtoms()`
  Returns the number of atoms on this patch.

- `int *getAtomList()`
  Returns the list of global indexes for atoms on this patch.

- `int id()`
  Returns the patch id of this patch.

- `void send_atom_reassignments()`
  Send out messages that contain atoms that have left the region of space owned by this patch and entered the region owned by a neighbor. A message is sent to every neighbor, even if the message says simply that there are no atoms to be transferred.

- `void receive_atom_reassignment(Message *msg)`
  Process the message `msg` that contains atoms that have entered the region of space owned by this patch from a neighboring patch.

- `void perform_rescale(BigReal factor)`
  Rescale atom velocities by multiplying by `factor`.

- `void patch_debug()`
  Debugging routine that dumps out the current status of the patch to the screen. This function is not normally called anywhere in the code, and exists solely for debugging purposes, hence the name ....

Private Functions::
These functions are only called within the `Patch` object itself, and they perform much of the work in the simulation.

- `void all_coordinate_init(Message *msg, int neighborId, int neighborIndex)`
  Initialization routine for messages received from neighbors who send this patch all of their atom coordinates. The forces for the current timestep are also calculated. The processing of this message includes passing the coordinates to all of the force objects and building the list of bonded coordinates that must be sent back to this neighbor. `msg` is the `Message` object containing the data, `neighborId` is the patch id of the neighbor who sent this message, and `neighborIndex` is the index of this neighbor into the `recvNeighbors` and other associated arrays. This function is only used during the first timestep of each cycle. During mid-cycle steps, the function `all_coordinate_force()` is used to calculate the forces without doing the initialization activities.

- `void all_coordinate_force(Message *msg, int neighborId, int neighborIndex)`
  Calculate forces with a neighbor that sends us all of their coordinates. `msg` is the message containing the coordinates, `neighborId` is the patch id of the neighbor that sent the message,

and `neighborIndex` is the index of this neighbor into the `recvNeighbors` and associated arrays. This routine is used during the mid-cycle timesteps. (i.e. timesteps after initialization has taken place.)

- `void bonded_coordinate_init(Message *msg, int neighborId)`
  Initialization routine for messages received from neighbors who send this patch only atom coordinates needed for bonded calculations. The forces for the current timestep are also calculated. The processing of this message includes the passing of the coordinate data to each of the bonded force objects. `msg` is the `Message` object that contains the data that was sent, and `neighborIndex` is the patch id of the neighbor that sent this message. This function is only used during the first timestep of each cycle. During mid-cycle steps, the function `bonded_coordinate_force()` is used to calculate the forces without doing the initialization activities.

- `void bonded_coordinate_force(Message *msg, int neighborId)`
  Calculate forces with a neighbor that sends this patch only bonded coordinates. `msg` is the message containing the coordinates and `neighborId` is the patch id of the neighbor who sent the message. This function is used during mid-cycle timesteps.

- `void prepare_bonded_coords(BondedWithNeighbors *bwn)`
  Gather the bonded coordinates that are to be sent to a neighbor. This function performs the loop described above that performs the gather operation necessary to send bonded coordinates to a neighbor. The `BondedWithNeighbors` structure that describes the coordinates that need to be sent are passed in `bwn`.

- `void local_force()`
  Calculate local interactions for the current timestep.

- `void local_init()`
  Initialize the local force interactions. This prepares all the force objects for calculating local interactions for this cycle and also calculates local forces for the current timestep. This function is used only during the first timestep in each cycle. The function `local_force()` is used to calculate forces during mid-cycle timesteps.

- `void add_forces(Message *msg)`
  Given a message `msg` that contains forces calculated by a neighboring patch that we sent all of our atom coordinates to, add these forces to the local forces for the current timestep.

- `BigReal get_kinetic_energy()`
  Calculate the kinetic energy for the current timestep. This function calculates the kinetic energy for the local atoms using the formula:

$$KE_{local} = \sum_{local\ atoms} \frac{1}{2} m v^2$$

- `int get_list_index(int value, const int *list, int numvalues)`
  This function searches for a value in a list of integers and returns the index of the value in the list. It assumes that the value will always be found, and it will terminate the program if a value is not found. It is used during atom reassignment to find the index of a neighbor

patch in the list of all neighbors. It is currently implemented as a binary search. `value` is the integer to be searched for, `list` is the list of integers to search, and `numvalues` is the number of integers in `list`.

- `void update_atom_info()`
  Perform all the actual work for atom reassignment. This function is the one that does all of the actual reassignment of values for atom reassignment. It is called once all the atom reassignment messages have been received from the neighbors of this patch. It is only called if atoms have entered or left this patch's region of space. It recreates the arrays `atoms`, `x`, `v`, and `f` to match the new number of atoms that reside on this patch.

5.3.31 PatchDistrib

Purpose:
The `PatchDistrib` object is used to determine and report the distribution of patches to processors. For the current implementation, it is responsible for determining what processor every patch on the system belongs to and distributing this information to the `PatchDistrib` object on every processor. Each object is then also responsible for returning the processor location of any patch. In order to be efficient, the `PatchDistrib` allows constant time access to patch information in two ways, by patch number and by grid coordinates. Grid coordinates identify the position of a patch in a set of grid coordinates where the origin is the lowest corner of the model.

Derived from:
None.

Files:
`PatchDistrib.C, PatchDistrib.h`

Constructor:
`PatchDistrib()`

Destructor:
`~PatchDistrib()`

Method of Use:
The only part of this class that is currently implemented is the creation of an initial distribution and the sending of this distribution from the client node to all the client nodes.

The creation of the initial patch distribution is done with the function `create_initial_distrib()` which is passed a `PDB` object that contains the initial positions of all the atoms. This function obtains the maximum and minimum corners of the molecule. It then determines how many patches it will take to cover these dimensions. Then, a layer of empty patches is added around all edges of the system to provide the buffer zone of empty patches that will be used. Patch structures are then created for each of these patches. Coordinates, grid positions, and patch numbers are then assigned to each patch. Hydrogen atoms are assigned to the same patch as the heavier atom to which they are attached. Also, the number of neighbors, the patch ids of the neighbors, and which neighbors to send and receive from are also determined. The patches to send and receive from are determined in a very simplistic way at the moment. The grid coordinates are compared in order from x to y to z. If the first coordinate that is not equal is found to be greater, than the patch is sent to. Otherwise, it is received from. Next, these patches are mapped to processors. A recursive coordinate bisection algorithm is used to map patches to processors. The `create_initial_distrib()` invokes the `RecBisection` object to accomplish this. If recursive bisection algorithm fails for some reason, then a VERY simplistic scheme is used where an equal number of patches are assigned to each processor in sequential patch number order (the function `simple_strip_division()`).

Now this distribution can be sent from the master process to the client processes. Once it is receive on each processor, the `PatchList` object can use this information to determine how many patches it is responsible for and it can allocate these `Patch` objects.

Next, since the PDB information was already parsed to determine the initial distribution, this information can now be used to send the initial coordinates and velocities to each `Patch` object. This isn't necessarily the most logical place for this to be done, but it turns out to be quite practical.

Public data:

- `int maxPatchNum` The biggest existing patch number.
- `int numPatches` Total number of patches int the system.
- `int patchDimension` Length of the side of a patch.

Functions:

- `void create_initial_distrib(PDB *)`
  Create an initial distribution of patches to processors using the initial positions of all the atoms
- `void send_Distrib(Communicate *)`
  Send a distribution from the master node to the client nodes
- `void receive_Distrib(Message *)`
  Receive a distribution from the master node
- `void send_initial_positions(Communicate *, PDB *)`
  Send out the initial atom coordinates and global atom indexes to each patch. These coordinates are obtained from the `PDB` object that is passed.
- `void send_initial_velocities(Communicate *, Vector *)`
  Send out the initial atom velocities to each patch. The velocities that are to be assigned to each atom are passed in via an array of Vectors.
- `int patch_node(int pnum)`
  Return the node that the `Patch` specified by `pnum` resides on.
- `int get_patch_id(int i, int j, int k)`
  Given the grid coordinates, return the patch number. This is intended to be used to determine a neighboring patch in a given direction.
- `void get_patch_positions(int pnum, int &i, int &j, int &k, Vector &origin)`
  Returns the patch coordinates for a patch number.
- `void get_num_atoms(int pnum)`
  Returns the number of atoms for the given patch.
- `void get_patch_origin(int pnum, Vector &origin)`
  Returns the origin of the given patch.

- `IntList *patches_for_node(int)`
  Given a node number, return an `IntList` object that contains the patch numbers assigned to that node. This is intended to be used by a given `Node` object to determine what patches belong to it.

- `int num_recv_for_patch(int pnum)`
  Given a patch number, return the number of neighbors that this patch will receive all coordinates from.

- `int num_send_for_patch(int pnum)`
  Given a patch number, return the number of neighbors that this patch will send all coordinates to.

- `void get_recv_patches(int pnum, int *plist)`
  The list of patch numbers that the patch `pnum` will receive all atoms from is returned in the array `plist`.

- `void get_send_patches(int pnum, int *plist)`
  The list of patch numbers that the patch `pnum` will send all its atoms to is returned in the array `plist`.

- `void get_FMA_cube(BigReal &boxsize, Vector &boxcenter)`
  Computes the box which contains all the patches, to be used in FMA evaluation.

- `void receiveChangeMsg(Message *msg)`
  Processes the patch migration messages produced by the load balance object.

5.3.32 PatchList

Purpose:
Manages and controls the set of Patch objects which reside on each processor node. The PatchList contains routines to add, move or delete patches, and contains the logic to complete one individual timestep. The Patch objects (section 5.3.30) then contain particular atoms and have the knowledge of how to calculate forces and integrate the equations of motion.

Derived From:
Nothing.

Files:
`PatchList.h, PatchList.C`

Constructor:
`PatchList(void)`

Destructor:
`~PatchList(void)`

Enumerations:

- `TSType : BEGIN_CYCLE, MID_CYCLE`

Defined Constants:
None yet.

Method of Use:
A single PatchList object exists on each processor node; it holds all the individual patches and loops through them to do a timestep calculation. It is created during the initial simulation initialization, before any timesteps are calculated, and deleted when the simulation is complete. When constructed, it contains no patches; these are added by the PatchConfig object, which calls specific routines within PatchList (see below) to add patches that should be on the node.

A node-level algorithm is responsible for determining when each timestep should be calculated; when a timestep is to be done, this algorithm calls `do_timestep(int tsType)` in the PatchList object. The argument `tsType` is a code to indicate the type of timestep this is to be; in the current design, there are only two types:

1. `tsType = BEGIN_CYCLE` : The very first timestep in a cycle of K steps. For this timestep, data may have to be regenerated for the PatchList and each patch due to movement of atoms, and redistribution/creating/deletion of patches.

2. `tsType = MID_CYCLE` : All other timesteps in a cycle other than the first; for these steps, the status of atoms and patches is taken to be static, and local data structures do not have to be regenerated.

When `do_timestep(int)` is called, the PatchList allows each Patch to calculate the forces on its atoms, and to integrate the equations of motion. The algorithm used to do this is:

1. If `tsType == BEGIN_CYCLE`, regenerate PatchList data structures due to changes in atom and Patch distribution.
2. `patchesDone = 0`.
3. For all patches on this node: call `patch.send_msgs(tsType)`.
4. While `patchesDone <` patches in PatchList:
   (a) If Message for a Patch available: call `patch.process_msg(msg)`.
   (b) increment patchesDone if `patch.process_msg(msg)` returned TRUE.
5. For all patches on this node: call `patch.report_data(tsType)`.

The Patch object is responsible for sending itself messages with commands to do such things as compute local forces, and to integrate the equations of motion once all forces are ready. The Communicate object must then return messages of a given tag with the following priority:

1. From a remote node.
2. From the local node.

Patches which send messages to another node do so by calling the routine `send_to_patch(Message *, patch, id)`. The argument `id` is a code used by the Patches to distinguish the contents of the Message; it is not the tag. A specific tag is used for all messages that are sent between Patch objects; this tag is defined in `common.h`.

Public data:

- `localIndexes`
  A global to local patch index translation table.

Functions:

- `void startup(IntList *)`
  Initialize the object. It takes an IntList that contains the patch ids that will be owned by this object and creates all the necessary data structures and initializes them.
- `void receive_initial_positions()`
  This function is used only during initialization. It is just a loop that waits for incoming position messages for each of the patches that it owns and passes the messages off to the appropriate patch object.
- `void receive_initial_velocities()`
  This function is used only during initialization and is the velocity equivalent to
  `void receive_initial_positions()`.

- `void send_nb_to_patches(int num_send, int *dest_patch, int timestep, int myId, int numAtoms, int *atoms,int *x, int *cycle_type)`
  Assembles a single message for each node which owns one or more destination patch. If only one patch on a node requires the atom positions, the same format and tag is used as in `send_to_patch`. If more than one patch requires the data, the Message sent has the following format:
  - one destination patch number for each patch on the remote node.
  - -1, to mark the end of the destination patch list.
  - timestep
  - `myId` (sending patch id)
  - `numAtoms` (`BEGIN_CYCLE` only)
  - global atom numbers (`BEGIN_CYCLE` only)
  - coordinates

  The message is unpacked on the receiving end, and local messages produced which follow to format produced by `send_to_patches`.

- `void send_to_patch(Message *msg, int patch, int id)`
  Sends the given Message to the given patch, encoding within the Message which patch the Message is for, the id, and the current timestep. The Message sent has the following format:
  - Original data in message.
  - id.
  - Sending patch number.
  - Destination patch number.
  - Timestep.

- `void patch_done(int pnum)`
  Tells `patch_list` that the indicated patch is done with the current timestep.

- `void do_timestep(int tsType)`
  Where the action is; this does one complete timestep, at the end of which the energies, coordinates, etc. are reported to the master node. The type of timestep to do is indicated by the argument.

- `void store_message(Message *msg, int tag)`
  Puts a message for a future timestep in a buffer, from where it can be retrieved when that timestep starts.

- `void updateFromChangeMsg(Message *msg)`
  Receives a patch migration message produced by the `LoadBalance` module and causes several actions to occur:
  1. Patches to be sent elsewhere are packed into messages, sent, and deleted.
  2. Each incoming patch creation message is received.

3. The new patches are created with the state in the creation message.

- `int numMyPatches(void)`
  Returns the number of patches supervised by this object.

Derived Classes:
None.

5.3.33 PDB

Purpose:
This is the class which manages the data in an entire XPLOR type PDB file. In the most general sense, it should organize the various data types present in the file. However, that generality is not needed. Thus, all this class does is read the ATOM and HETATM records via the PDBAtomRecord and PDBHetAtm classes, respectively. The generality (and complexity) remains in the code because it was taken from a project to read in and manipulate all the PDB records.

The atom information is accessed as an array. There are several ways to search the information in the PDB class. All these search function are named `find_atom_*`, where the "*" can be one of many possible search criterion. The result of a search is an `IntList` pointer which contains the indicies to all the atoms found. Memory is allocated for this list, so it must be deleted after it is no longer needed.

Derived From:
Nothing.

Files:
`PDB.C, PDB.h, IntList.h`

Constructor:
`PDB( const char *pdbfilename)` where `pdbfilename` is the file name of the input XPLOR type PDB file. The full description of a standard PDB file is available via anonymous ftp to pdb.pdb.bnl.gov, in the file /pub/format.desc.ps. There are two differences between this standard format and the XPLOR variant, which are explained in `PDBAtom`.

The constructor reads the file and stores the ATOM and HETATM records into one linked list. After all the data is read, the linked list is converted into an array, in order to increase access speed.

Destructor:
`PDB( void)`

Method of Use:

```
PDB pdbinput(”pti.pdb”);
if (pdbinput.numatoms() == 0 )
NAMD_die(”There were no atom records in the input file.”);
IntList *search = pdbinput.find_atom_name(”CA”);
cout << ”There are ” << search -> num() << ”alpha carbons./n”;
cout << ”The fifth is atom number ” <<
pdbinput.atom(search[5])-¿serialnumber() << ”./n”;
delete search;
```

Typedefs:

`PDBAtomList` This contains two elements, `data` is a PDBAtom * to the atom (which is either an ATOM or HETATM record) and `next` is a `PDBAtomList *` to the next element of the list. The list is NULL terminated. This is used internally to the class to maintain the list of all atoms read from the PDB file before it is converted into an array.

Functions:

- `void write(const char *outfilename, const char *commentline = NULL)`
  Writes the coordinates to a file.
- `int num_atoms( void)`
  Returns the total number of atoms read.
- `IntList *find_atom_serialnumber(int serialnumber)`
  Returns indices to atoms which match the serial number
- `IntList *find_atom_name(const char *name)`
  Returns indicies to atoms which match the name (as in "CA").
- `IntList *find_atom_alternatelocation(const char *alternatelocation)`
  Returns indicies to atoms which match the alternate location field.
- `IntList *find_atom_residuename(const char *residuename)`
  Returns indicies to atoms which match the residue name (as in "ALA").
- `IntList *find_atom_chain(const char *chain)`
  Returns indicies to atoms which match the chain identifier.
- `IntList *find_atom_residueseq( int residueseq)`
  Returns indicies to atoms which match the residue sequence number.
- `IntList *find_atom_insertioncode( const char *insertioncode)`
  Returns indicies to atoms which match the insertion code.
- `IntList *find_atom_segmentname( const char *segmentname)`
  Returns indicies to atoms which match the segment name - this is the XPLOR extension to the PDB file format.
- `IntList *find_atom(const char *name=NULL,const char *residue = NULL, int residueseq = -1, const char *segment = NULL)`
  This provides the ability to search based on several criterion at once. If a field contains its default value, it will not be used in the search
- `PDBAtom *atom(int place)`
  Returns the pointer to atom number "place". Thus, `atom(1)` is the 2nd element present.
- `PDBAtomList *atoms(void )`
  Returns the head of the array. Messing with this may cause nastiness to occur.

- `IntList *find_atoms_in_region( Real x1, Real y1, Real z1, Real x2, Real y2, Real z2 )`
  Returns indicies to atoms which lie in a certain region of space. Assuming that $x1 < x2$, $y1 < y2$, $\ldots$, then an atom at position (x, y, z) is in the region iff $x1 <= x < x2$, $y1 <= y <= y2$, $\ldots$. The function will reorder the coordinates so that the $x1 < x2$ assumption is true.

- `void find_extremes(Vector *, Vector *)`
  Returns two vectors, one with the minimum x, y, and z coordinates in the system and the other with the maximum x, y, and z coordinates in the system.

- `void set_all_positions(Vector *x)`
  Resets all the positions in the PDB object.

- `void get_all_positions(Vector *x)`
  Puts all the positions in the PDB object into `x`.

5.3.34 PDBAtom, PDBAtomRecord, PDBHetatm

Purpose:
These are the classes which store the data for the two PDB atom coordinate records. There are two of these types of records in the PDB file, the ATOM record and the HETATM record. The fields are the same in both except for the name. Columns 1-6 of an ATOM record are "ATOM " and columns 1-6 of a HETATM record are "HETATM". Otherwise the fields are identical. The `PDBAtom` class keeps track of the data which are the same and uses the function `type()` (derived from `PDBData`) to determine the first 6 columns.

There are some differences between the PDB atom formats used by NAMD and the one given by the Brookhaven Protein Data Bank. This is because NAMD uses the X-PLOR variant of that format. The PDB says that the residue name field is 3 characters long whereas NAMD accepts 4 character names. The PDB uses the 1 character chain identifier. X-PLOR instead uses columns 73 to 76 to define the "segment name" (see XPLOR 3.1 manual, p 104). This PDBAtom class will read both the chain and the segment identifier names.

In addition, there is another PDB variant which this class will read and write. There are many quick manipulations which can be done with Unix tools such as awk and perl. These tools work best with field based data. However, the PDB is a column based data file. This class provides the ability to convert to a field based format, do some manipulations, and convert back to a column based format. NAMD does not use this alternate format for anything.

Derived From:
PDBData

Files:
`PDBData.C, PDBData.h`

Enumerations:

- `Start`
  (STYPE=1,SSERIAL=7, SNAME=13, SALT=17, SRESNAME=18, SCHAIN=22, SRESSEQ=23, SINSERT=27, SX=31, SY=39, SZ=47, SOCC=55, STEMPF=61, SFOOT=68, SSEGNAME=73) the starting column of each data element
- `Length`
  (LTYPE=6, LSERIAL=5, LNAME=4, LALT=1, LRESNAME=4, LCHAIN=1, LRESSEQ=4, LINSERT=1, LCOOR=8, LCOORPREC=3, LOCC=6, LOCCPREC=2, LTEMPF=6, LTEMPF-PREC=2, LFOOT=3, LSEGNAME=4) – the length of each data element
- `PDBPossibleAtoms`
  (USE_ATOM = ATOM, USE_HETATM = HETATM) – specifies if the element is to be an ATOM or HETATM record

Contructor:
`PDBAtom(char *dataline, PDBPossibleAtoms whichatom)` Given the line as presented in the PDB file, convert the information to either an ATOM or HETATM record.

`PDBAtomRecord(char *dataline)` Given the line from the PDB file, tell it parent class, `PDBAtom`, to make an ATOM record.

`PDBHetatm(char *dataline)` Given the line from the PDB file, tell it parent class, `PDBAtom`, to make a HETATM record. *Destructor*:

```
PDBAtom( void)
    PDBAtomRecord( void)
    PDBHetatm( void)
```

Functions:

- `void parse( const char *s)`
  Given a line in the PDB format, extract the coordinate information
- `sprint(char *s, PDBFormatStyle usestyle = COLUMNS)`
  Print the current data in one of the two supported PDB variants
- `int serialnumber( void)` read or change the atom serial number
- `void serialnumber( int newserialnumber)`
- `const char*name( void)` read or change the atom name
- `void name( const char *newname)`
- `const char*alternatelocation( void)` read or change the alternate location
- `void alternatelocation( const char *newalternatelocation)`
- `const char*residuename( void)` read or change the residue name
- `void residuename( const char *newresiduename)`
- `const char*chain( void)` read or change the chain identifier
- `void chain( const char *newchain)`
- `int residueseq( void)` read or change the residue sequence number
- `void residueseq( int newresidueseq)`
- `const char*insertioncode( void)` read or change the insertion code.
- `void insertioncode( const char *newinsertioncode)`
- `Real xcoor( void)` read or change one of the x, y, or z coordinates
- `void xcoor( Real newxcoor)`
- `Real ycoor( void)`
- `void ycoor( Real newycoor)`

- `Real zcoor( void)`
- `void zcoor( Real newzcoor)`
- `const Real *coordinates( void)` read or change all three coordinates
- `void coordinates(const Real *newcoordinates)`
- `Real occupancy( void)` read or change the occupancy
- `void occupancy( Real newoccupancy)`
- `Real temperaturefactor( void)` read or change the temperature factor
- `void temperaturefactor( Real newtemperaturefactor)`
- `int footnote( void)` read or change the footnote number
- `void footnote( int newfootnote)`
- `const char*segmentname( void)` read or change the segment name
- `void segmentname( const char *newsegmentname)`

5.3.35 PDBData, PDBUnknown

Purpose:
Each line of a PDB file is a data record. There are many different types of records, from TITLE to MATRIX to ATOM and each record is maintained by a seperate class, all of which are derived from PDBData. This class contains an integer which describes which type of PDB data record it is. This allows other pieces of code to recast the base class into the proper subclass. This class also contains some protected functions which can read and write to specific columns in a string, since the PDB file is a column based file.

The most trivial example of a subclass is `PDBUnknown`, which is constructed with the string containing the data record. All it does is copy and save that string so that it may be printed later.

Global Function:
`PDBData *new_PDBData(const char *data)`
All of the derived classes are constructed with a char * containing the line from the PDB file. Since the parent class cannot create the appropriate child (in C++), there must be a helper function which knows which child to create. For the classes derived from `PDBData`, the appropriate helper function is `new_PDBDdata`. It knows how to look as the “data” string to figure out which child to create. It then calls the constructor for that class and returns the pointer.

Derived From:
Nothing.

Files:
`PDBData.C, PDBData.h`

Enumerations:

- `PDBType` (HEADER, OBSLTE, COMPND, SOURCE, EXPDTA, AUTHOR, REVDAT, SPRSDE, JRNL, REMARK, SEQRES, FTNOTE, HET, FORMUL, HELIX, SHEET, TURN, SSBOND, SITE, CRYST1, ORIGX, SCALE, MTRIX, TVECT, MODEL, ATOM, HETATM, SIGATM, ANISOU, SIGUIJ, TER, ENDMDL, CONECT, MASTER, END, UNKNOWN) – the different possible PDB records
- `PDBFormatStyle` (COLUMNS, FIELDS)
  There are two ways to print the data. The default is the column based format of the PDB, where a record is, for example, the integer between columns 35 and 38. The other format is FIELDS, in which each data element is seperated by a space and unknown elements are replaced by a '#'. This is useful for UNIX text manipulation tools like awk and perl. /progname/ only uses the COLUMNS format.

Contructor:
`PDBData(PDBType newtype)` where type is the PDB record type of this PDB data. *Destructor*:

`PDBData( void)`

Functions:

- `PDBType type( void)`
  Returns the type of data which this class contains
- `static void scan( const char *data, int len, int start, int size, int *ans, int defalt)`
  Given a "data"string of length "len", start at position "start" and read the next "size" characters to read an integer and put the value into "ans". If the field is blank, use "defalt" instead.
- `static void scan( const char *data, int len, int start, int size, Real *ans, Real defalt)`
  Same as above, but with Real data.
- `static void scan( const char *data, int len, int start, int size, char *ans)`
  Same as above, but with a char *, and there is no default value.
- `static void field( const char *data, int fld, char *result)`
  Read in field number "fld" from the string "data" and put the information into "result".
- `static void sprintcol( char *s, int start, int len, const char *val)`
  Print up to "len" characters of "val" into the string "s" starting at position "start".
- `static void sprintcol( char *s, int start, int len, int val)`
  The same, but with the integer "val".
- `static void sprintcol( char *s, int start, int len, int prec, Real val)`
  The same, but with the Real "val", with "prec" digits of precision.
- `virtual void sprint( char *s, PDBFormatStyle usestyle = COLUMNS)`
  Print all the data concerning this record into the string "s" using the given format style.

5.3.36 RecBisection

Purpose:
The `RecBisection` object is used to partition the the computational domain (patches) into p regions where p is the number of processors. This object is a friend of `PatchDistrib` object and accesses directly to its private data structure that holds the map of patches. The partitioning algorithm has three steps: (a) the recursive coordinate bisection method is used to determine a temporary partitioning into 3D rectangular prisms where the prisms have approximately equal loads. The load of a prism is simply the sum of loads introduced by each patch in the prism. The compute_patch_load() function determines the load of each patch; (b) After initial partitioning, the boundaries between regions are modified to improve the communication; (c) finally, the force calculation directions are revisited to further improve the computational load of regions. At this moment, only the step (a) is implemented.

Derived from:
None.

Files:
`PatchDistrib.C, PatchDistrib.h`

Enumerations:
directions (XDIR=0,YDIR,ZDIR)

Constructor:
`RecBisection()`

Destructor:
`~RecBisection()`

Method of Use:
The partitioning is performed by invoking the function `partition()`. This function is called by the `PatchDistrib` object after the computational domain is divided into patches. Then, `RecBisection` accesses the private data of `PatchDistrib` (for patch information), performs partitioning, and assigns partitions to processors.

Public Functions:

- `int partition(int *dest_arr)`
  This is the function invoked from outside to perform partitioning.

Private Functions:

- `void compute_patch_load()`
  Compute the load introduced by each patch.

- `void rec_divide(int n, const Partition&P)`
  Recursively divide a given partition P into n subpartitions.

- `void assignNodes()`
  Assign the partions and patches to the processors.

- `void assign_nodes_arr(int *dest_array)`
  The partitioning is evaluated, but the results are assigned to an array instead of updating the PatchDistrib structure directly.

- `void refine_edges()`
  to be implemented.

- `void refine_boundaries()`
  to be implemented.

- `void refine_surface()`
  to be implemented.

- `int prev_better(float prev, float current, float load1)`
  return true if teh previous bisection point is better then the current one.

5.3.37 RigidHBonds

Purpose:
The RigidHBonds class is used to implement rigid bonds to hydrogens. There is a `RigidHBonds` object present in each patch. This object performs necessary rigid-bond calculations for the atoms within that patch. Note that the rigid-bond calculations involve only the bonds to hydrogens. Since hydrogen atoms are always assigned to the same patch as their parent atom, this class does not require atom information from neighboring patches, i.e., all calculations are local to the patch.

If rigid bond calculations are turned on by the user, the `RigidHBonds` object initializes a table of rigid-bonds present in the patch at the beginning of each cycle. During the cycle this list will not change. The force calculations continue as they are until the integration step (except the bonds with hydrogen and water molecules will be excluded from bond and angle calculations). After the positions and velocities are updated by the integrator, the RigidHBonds object adjusts the positions of those atoms involved in rigid bonds. Similarly the velocities of those atoms are adjusted by Rattle.

In order to save some memory, the positions before the integration (`RigidHBonds` object needs this) is saved in the forces array instead of creating a separate array. This is a reminder in case you need to modify the code.

Derived From:
Nothing.

Files:
`RigidHBonds.h, RigidHBonds.C`

Constructor:
`RigidHBonds()`

Destructor:
`~RigidHBonds()`

Method of Use:
Each patch creates a `RigidHBonds` object to do rigid calculations for the atoms in that patch. There are three functions that Patch class invokes: a) `init()` which initializes the table of rigid bonds to be calculated, b) `force()` which does the calculations (i.e., adjusting the bond lengths, and c) `reset_lists()` which invalidates the rigid-bond table at the end of the cycle (due to atom movements).

The rigid-bond calculations are done by four internal functions: `apply_rigid1H()`, `apply_rigid2H()`, `apply_rigid3H()`, and `apply_rigidWater()`. The function `force()` invokes appropriate calculation for each rigid-group. Each apply_rigid function calculates the new bond lengths and adjusts the position of those atoms directly. Note that the calculation functions need to be optimized to increase the sequential performance.

Public Functions:

- `void init(PatchList *, int[], int)`

- `void force(int [], Vector [], Vector[], Vector[])`
- `void reset_lists()`

Private Functions:

- `Bool apply_rigid1H(Vector [], int, Real [], int [])`
- `Bool apply_rigid2H(Vector [], int, Real [], int [])`
- `Bool apply_rigid3H(Vector [], int, Real [], int [])`
- `Bool apply_rigidWater(Vector [], Vector [], int, Real [], Real, int [])`

5.3.38 Rattle

Purpose:
The Rattle class implements rigid velocities for hydrogens. There is a `Rattle` object present in each patch. This object performs necessary rigid-velocity calculations for the atoms within that patch. Note that the rigid-velocity calculations involve only the bonds to hydrogens. Since hydrogen atoms are always assigned to the same patch as their parent atom, this class does not require atom information from neighboring patches, i.e., all calculations are local to the patch.

The Rattle class works in conjunction with the RigidHBond class. While the RigidHBond class restricts atom positions, the Rattle class restricts atom velocities.

If rigid velocity calculations are turned on by the user, the `Rattle` object initializes a table of rigid-velocities present in the patch at the beginning of each cycle. During the cycle this list will not change. The force calculations continue as is until the integration step (except the bonds with hydrogen and water molecules will be excluded from bond and angle calculations). After the positions and velocities are updated by the integrator, the Rattle object adjusts the velocities of those atoms involved in rigid bonds. Similarly, the positions of those atoms are adjusted by RigidHBonds.

In order to save memory, the positions before the integration (required by the `Rattle` object) are saved in the forces array instead of creating a separate array. This is a reminder in case you need to modify the code.

The basic functionality of Rattle is to modify the atom velocities, V, of each group. This is done by manipulating a bond-atom matrix, χ_x, and a diagonal matrix of masses, M. The overall function is $V^* = V - M^{-1}\chi_x^{\mathrm{t}}(\chi_x M^{-1}\chi_x^{\mathrm{t}})^{-1}\chi_x V$. The kth element of the column vector χ is one-half the length squared of the kth bond length.

For waters, the matrix M is a 9×9 diagonal matrix of the masses. The χ^{t} matrix (transpose of χ) is a 9×3 table relating atoms with bonds. Each row in the table corresponds to each atom's x, y, z position. The columns relate to the bonds. The final vector, V^*, is a concatenation of each atom's velocities.

Derived From:
Nothing.

Files:
`Rattle.h, Rattle.C`

Constructor:
`Rattle()`

Destructor:
`~Rattle()`

Method of Use:
Each patch creates a `Rattle` objects to do rigid calculations for the atoms in that patch. There are three functions that Patch class invokes: (a) `init()` which initializes the table of rigid bonds to be calculated, (b) `force()` which does the calculations (i.e., adjusting the velocities, and (c)

`reset_lists()` which invalidates the rigid-velocity table at the end of the cycle (due to atom movements).

The rigid-velocity calculations are done by four internal functions: `apply_rigid1H()`, `apply_rigid2H()`, `apply_rigid3H()`, and `apply_rigidWater()`. The function `force()` invokes appropriate calculation for each rigid-group. Each apply_rigid function calculates the new velocities and adjusts the velocities of those atoms directly.

Public Functions:

- `void init(PatchList *, int[], int)`
  Initialize the table of rigid bonds. This is identical to the table generated by `RigidHBonds::init()`. A future improvement may combine these two tables.
- `void force(int [], Vector [], Vector[], Vector[])`
  Perform the rigid-velocity calculations.
- `void reset_lists()`
  Resets the table of rigid bonds at the end of a cycle. This method is used to prevent invalid entries due to atom movement.

Private Functions:

- `Bool apply_rigid1H(Vector [], Vector[], int, int [])`
  Apply `Rattle` to an atom with a single hydrogen bond.
- `Bool apply_rigid2H(Vector [], Vector[], int, int [])`
  Apply `Rattle` to an atom with two hydrogen bonds.
- `Bool apply_rigid3H(Vector [], Vector[], int, int [])`
  Apply `Rattle` to an atom with three hydrogen bonds.
- `Bool apply_rigidWater(Vector [], Vector[], int, Real, int [])`
  Apply `Rattle` to a water. This is different than `apply_rigid3H` since there is an interaction between the two hydrogens.

5.3.39 RigidHData

Purpose:
The class builds a database of atom-groups that are subject to rigid-bond calculations. There are four such groups: a non-hydrogen atom with bonds to 1) one hydrogen (1H pattern), 2) two hydrogens (2H pattern), 3) three hydrogens (3H pattern), 4) and finally water molecule. In order to determine these groups, this class uses the list of bonds, angles, and impropers from Molecule class.

Derived From:
Nothing.

Files:
`RigidHData.h, RigidHData.C`

Constructor:
`RigidHData()`
The constructor initializes some private data members.

Destructor:
`~RigidHData()`

The destructor frees the memory allocated for various tables.
Method of Use:
The Node object (on each processor) calls the *setup()* function to initialize the rigid-bonds database. *setup* takes one parameter: *number of degrees of freedom.* This parameter is adjusted properly depending on the rigid bonds.
Public Functions:

- `void setup(int &numDegFreedom)`
  setup takes one parameter: *number of degrees of freedom.* This parameter is adjusted properly depending on the rigid-bond group found. The *setup* function first inspects each bond. If the bond is between one hydrogen and one non-hydrogen atom, then the bond is entered into the rigid-1H table. Note that these bonds will be exluded from the bond-force calculations (`BondForce` class checks if a bond is subject to rigid-bonds calculations, if so, the bonded interaction is skipped).

  Similarly, the list of angles are searched to find the rigid-2H atom groups and water molecules. Note that the angle calculations (`AngleForce` class) will skip the calculations of water molecules but the angle interactions for rigid-2H groups will be performed. There might be rigid-2H patterns which are not in the angle list. We have to find these atom-groups too. Once rigid-2H and water molecules are determined, the rigid-1H database is modified to eliminate the bonds that are already covered by the rigid-2H and water molecules. Finally, *setup* function determines rigid-3H groups and eliminates any rigid-2H or rigind-1H patterns already covered by the rigid3H groups.
- the rest of the functions are for accessing various information in the database.

5.3.40 Timer

Purpose:
The Timer class is used for timing various aspects of NAMDand is modeled after the CM5 implementation of CMMD_node_timers. Each Timer object is an accumulator that accumulates time whenever started. A Timer can be started and stopped multiple times, accumulating the total time from each cycle. Timer's can also be cleared to set the accumulated time to zero. Each Timer tracks clock time, user cpu time, system cpu time, and total cpu time.

On the HPs, these functions are based on the library routines `times()` and `gettimeofday()`.

Derived From:
Nothing.

Files:
`Timer.h, Timer.C`

Constructor:
`Timer()`

Destructor:
`~Timer(void)`

Method of Use:
The functions `start()`, `stop()`, and `clear()` are used to start, stop and clear a timer. The functions `clock_time()`, `cpu_time()`, `user_time()`, and `system_time()` are used to report the current values of a timer.

As example, to time both the total time to complete a loop and each iteration of the loop, the following code could be used:

```
> Timer total_time;
  Timer iter_time;

  total_time.start();
  while ( .  .  .)
  {
     iter_time.start();
     .  .  .
     iter_time.stop();
     printf("ITERATION TOOK %f seconds\n", iter_time.clock_time());
     iter_time.clear();
  }
  total_time.stop();
  printf("TOTAL TIME %f seconds\n", total_time.clock_time());
```

Functions:

- `void clear()`
  Clear the accumulated time for the Timer
- `void start()`
  Start the Timer accumulating time
- `void stop()`
  Stop the Timer from accumulating time
- `float clock_time()`
  Get the accumulated clock time for the Timer
- `float user_time()`
  Get the accumulated user CPU time
- `float system_time()`
  Get the accumulated system CPU time
- `float cpu_time()`
  Get the accumulated CPU time. This is `user_time()` + `system_time()`

5.3.41 Vector

Purpose:
This allows us to manipulate vectors as if they are standard data types. A vector is something which contains three "BigReal" values; x, y, and z. There are two ways to use Vectors. The most natural is through operator overloading, which allows us to say $\vec{v}_3 = \vec{v}_1 + \vec{v}_2$. However, this means that multiple copies may occur needlessly, so we provide an alternate means of doing these function. In essence, these are "two operand function". For example, `v1.add(v2)` will add `v2` to `v1`.

Derived From:
Nothing.

Files:
Vector.h, common.h

Constructor:

- `Vector( void)` by default, create a 0 vector
- `Vector( const Vector &v2)` copy constructor, Vector v = another_vector
- `Vector( BigReal x, BigReal y, BigReal z)` create a vector from its basic elements

Destructor:
`Vector( void)`

Method of Use:

```
Vector v1(1.1,2.2, 3.3); // create some vectors
cout << "v1.x == " << v1.x << '/n';
Vector v2(-1, 55, 32.1);
Vector v3(v1+2*v2);
cout << v1 << " " << v2 << " " << v3 << '/n';
Vector v4;
v4 = v3*5 - v2/4;
cout << cross(v3, v4) << '/n';
v3 = (v1 += v2) / -4.25;
v1.sub(v2);
v4.mult(3.14);
cout << v3.length() << " " << v2.dot(v1) << '/n';
v2 = v4.unit(); // return the vector along v2 of unit length
```

Operators:
These operators are provided to make working with Vectors look and feel like you are just working with real numbers. This makes the code cleaner and more natural.

- `friend int operator == (const Vector &v1, const Vector &v2)`
  compare two vectors, returns TRUE if they are the same
- `friend int operator != (const Vector &v1, const Vector &v2)`
  compare two vectors, returns TRUE of they are not the same
- `friend Vector operator+(const Vector &v1, const Vector &v2)`
  returns the Vector which is $\vec{v}_1 + \vec{v}_2$
- `friend Vector operator-(const Vector &v1, const Vector &v2)`
  returns the Vector which is $v1 - v2$
- `friend Real operator*(const Vector &v1, const Vector &v2)`
  returns the dot product of v1 and v2, which is $\vec{v}_1 \cdot \vec{v}_2$
- `friend Vector operator*(const Real &f, const Vector &v1)`
  returns the Vector which is $f \times \vec{v1}$
- `friend Vector operator*(const Vector &v1, const Real &f)`
  returns the Vector which is $\vec{v1} \times f$
- `friend Vector operator/(const Vector &v1, const Real &f)`
  returns the Vector which is $\vec{v1}\ f$
- `friend Vector cross(const Vector &v1, const Vector &v2)`
  returns the Vector cross product, which is $\vec{v}_1 \times \vec{v}_2$
- `friend Vector cross(const Real &k, const Vector &v1, const Vector &v2)`
  returns the scaled Vector cross product, which is $k \times \vec{v}_1 \times \vec{v}_2$
- `friend Vector cross(const BigReal &k, const Vector &v1, const Vector &v2)`
  returns the scaled Vector cross product, which is $k \times \vec{v}_1 \times \vec{v}_2$
- `friend ostream& operator<<(ostream& strm, const Vector &v1)`
  provides a means by which to print Vectors using streams. There is no corresponding input function

Functions:
These functions are not as versatile as friend functions above, but they should be faster as they don't need a copy.

- `Real &operator[](int i)` Lets us pretend that $\vec{V}_x$ =V[0], y is element 1, and z is 2. All other values cause NAMD to exit with an error.
- `Vector& operator+=(const Vector &v)` add $\vec{V}$ to this Vector
- `Vector& operator-=(const Vector &v)` subtract $\vec{V}$ from this Vector
- `void add(const Vector &v)` add $\vec{V}$ to this Vector, but doesn't return anything
- `void add_const(BigReal f)` add a scalar to each Vector element

- `void sub(const Vector &v)` subtract $\vec{V}$ from this Vector, but doesn't return anything
- `Real length( void)` returns the length (2-norm) of this Vector
- `Vector unit()` returns the Vector which in the same direction as this Vector, but is of unit length
- `void cross(const Vector &v)` replace this Vector by the cross product of this Vector with v
- `void mult(BigReal f)` multiply each Vector element by a scalar
- `void div(BigReal f)` divide each Vector element by a scalar
- `BigReal dot(const Vector &v)` compute and return the dot product of this Vector with v
- `Bool set(const char *s)` Set the vector values from a string, which is in the form "x y z" or "x, y, z".