Parallellizing molecular dynamics simulations on the Playstation multicore processor

Techincal Report, January 2007

Written for 6.189 - Multicore Programming Primer

Grigore D. Pintilie

Elecrical Engineering and Computer Science, MIT

Abstract

Molecular dynamics simulations are increasingly used to uncover how biologically relevant molecules such as proteins attain their 3-dimensional structure, and how this structure influences their function. At the core of such simulations lie a simple representation of atoms as spherical objects connected by springs which represent covalent bonds. In addition atoms also influence one another through repulsive van Der Waals forces, and attractive or repulsive electrostatic forces. This simplistic model becomes very computationally demanding as systems increase in size. To simulate systems that are of any biological significance, parallel computation is typically employed. The challenge in making such parallelizations efficient is the communication of data amongst the many processors involved. In this work, various parallelization schemes are considered - force, system, and spatial decomposition. The force decomposition scheme is implemented and tested on the multi-core chip.

1. Introduction to Molecular Dynamics

The fundamental step in a molecular dynamics (MD) simulation is the computation of forces on all the atoms in the systems. The force on an atom can be written as the derivative of a potential energy function, which includes bonded and non-bonded terms:

$$E_p(\vec{x}) = E_{bonded} + E_{non-bonded}$$

The energy function is written as a function of all the atom positions, here written in the form of a vector, $\vec{x}$. The bonded term in the above equation can be split into three separate (and independent) energy terms:

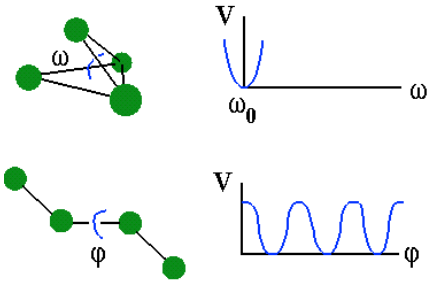
$$E_{bonded} = E_{bonds} + E_{angles} + E_{dihedral} + E_{improper}$$

The first two energy terms in the above equation are most commonly defined in terms of simple harmonic potentials, with the energy quadratically related to the deviation of the distance, l, or angle, θ, from the equilibrium values, l_0 and θ_0 respectively:

$$E_{bonds} = \sum_{bonds} k_b \left(l - l_0\right)^2$$

$$E_{angles} = \sum_{angles} k_\theta \left(\theta - \theta_0\right)^2$$

The bonds energy is computed for any two atoms that are covalently bonded, and the angle energy is computed for any two covalent bonds that share one atom (thus involving in all 3 atoms). The third and fourth bonded energy terms involve any four atoms that are covalently bonded to one another through 3 covalent bonds. The dihedral angle, φ, and the improper angle, ω, are illustrated below:



The illustration above also shows the shapes of the potential functions (V), which can be expressed mathematically as follows:

$$E_{improper} = \sum_{impropers} k_\omega \left(\omega - \omega_0\right)^2$$

$$E_{dihedral} = \sum_{dihedrals} \sum_{i} k_{\phi,i} \left[1 + \cos\left(n_i \phi_i + \delta_i\right)\right]$$

Thus the improper energy function also has a harmonic form, whereas the dihedral energy function is expressed in terms of cosine functions, each having a different frequency, as determined by n_i, and a phase shift of δ_i.

Now moving on to the non-bonded energy term, this can be decomposed into two separate terms:

$$E_{non-bonded} = E_{van-der-Waals} + E_{electrostatic}$$

The van der Waals term is computed for any two atoms, *i* and *j*, which are not directly bonded to one another, and is written empirically as:

$$E_{van-der-Waals} = \sum_i \sum_{j \neq i} 4\varepsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right]$$

The electrostatic energy term is based on the Coulombic potential function:

$$E_{electrostatic} = \sum_i \sum_{j \neq i} \frac{q_i q_j}{4\pi\varepsilon r_{ij}}$$

In all of the above equations, several constants or parameters need to be specified, and this set of parameters are generally referred to as the 'force field'. Example of existing force fields are CHARMM[1], AMBER[2], GROMACS[3], and OPLS[4]. In this work, the CHARMM force field was used.

The forces are computed for every atom by differentiating the energy function:

$$\vec{f}(t) = -\frac{\partial}{\partial \vec{x}} E_p(\vec{x})$$

The simulation then simply involves using Newton's formula for motion to get the acceleration of every atom:

$$\vec{f}(t) = M\vec{a}(t)$$

Instead of integrating acceleration to get velocity and once again to get position, the Verlet integration scheme is used:

$$\vec{v}\left(t + \frac{1}{2}\Delta t\right) = v\left(t - \frac{1}{2}\Delta t\right) + \Delta t \frac{\vec{f}(t)}{m}$$

$$\vec{x}(t + \Delta t) = \vec{x}(t) + \Delta t \cdot \vec{v}\left(t + \frac{1}{2}\Delta t\right)$$

This integration scheme has the desirable property that, given the step size is small enough, the total energy in the system does not increase or decrease as the simulation goes on. The step size is set

[1] MacKerell, Jr. AD, et al. (1998). "All-atom empirical potential for molecular modeling and dynamics studies of proteins". J Phys Chem B 102: 3586-3616

[2] Duan et al. (2003) "A point-charge force field for molecular mechanics simulations of proteins based on condensed-phase quantum mechanical calculations" Journal of Computational Chemistry Vol. 24, Issue 16. Pages 1999-2012.

[3] Van Der Spoel D, Lindahl E, Hess B, Groenhof G, Mark AE, Berendsen HJ (2005). "GROMACS: fast, flexible, and free". J Comput Chem 26 (16): 1701–18

[4] W. L. Jorgensen and J. Tirado-Rives. The OPLS Force Field for Proteins. Energy Minimizations for Crystals of Cyclic Peptides and Crambin. J. Am. Chem. Soc. 1988, 110, 1657-1666

based on the highest vibrational frequency in the system, and this is the vibration of a hydrogen atom covalently bonded to any other heavier atom such as carbon or oxygen. A step of 1fs is sufficient to capture this vibration; with any larger a step size the simulation can easily become unstable. If hydrogen atoms are not included in the simulation or are not simulated explicitly (e.g. using a relaxation algorithm), then a step size of 2fs can be used.

Based on the thermodynamic theory that, at equilibrium, every degree of freedom has a certain amount of energy, this being $\frac{1}{2}k_B T$. The total number of degrees of freedom, N_F, is the number of atoms in the system times 3 for each dimension of space. Thus:

$$\langle E_k \rangle = \frac{1}{2}\sum_{i=1}^{3N} m_i v_i^2 = \frac{1}{2}N_F k_B T$$

The formula above can be used to calculate the temperature, T, at every step in the simulation, and also to set the initial velocities. The initial velocity for any one atom is picked at random from a Gaussian distribution.

To make the simulation more physically realistic, the effect of an infinite water bath at constant temperature must be considered. One effect of this water bath is that very energetic particles are eventually slowed down as their energy dissipates into the water bath; one way to account for this is to add a drag term, in effect adding a force opposite in direction to velocity. A second effect of the water bath is that random or thermal noise is constantly stirring up the system. These two terms are typically added to the equation of motion as given below, an equation that is know as the Langevin equation:

$$\vec{f}(t) = -\nabla E_p(\vec{x}) - \gamma M\vec{v} + R(t)$$

Here, M was introduced as the mass matrix, or a matrix where the diagonal elements are the mass of each atom. The random function must be picked such that the average over time is zero, each random vector from time step to time step is completely independent of one another, and the magnitude of the random vector at each time step is proportional to the temperature being maintained:

$\langle R(t) \rangle = 0$, and $\langle R(t), R(t')^T \rangle = 2\gamma k_B T M \delta(t - t')$

One very important effect of this equation is that the simulation will maintain a constant temperature as it goes on, regardless of any artificial forces that might be added during the simulation.

2. Parallelizing the MD Computation

In parallelizing any application, two main issues are of high importance: how is the workload split amongst many processors, and how is the data shared amongst processors. Data communication is a vital consideration because each processor can access its own memory space efficiently, however it

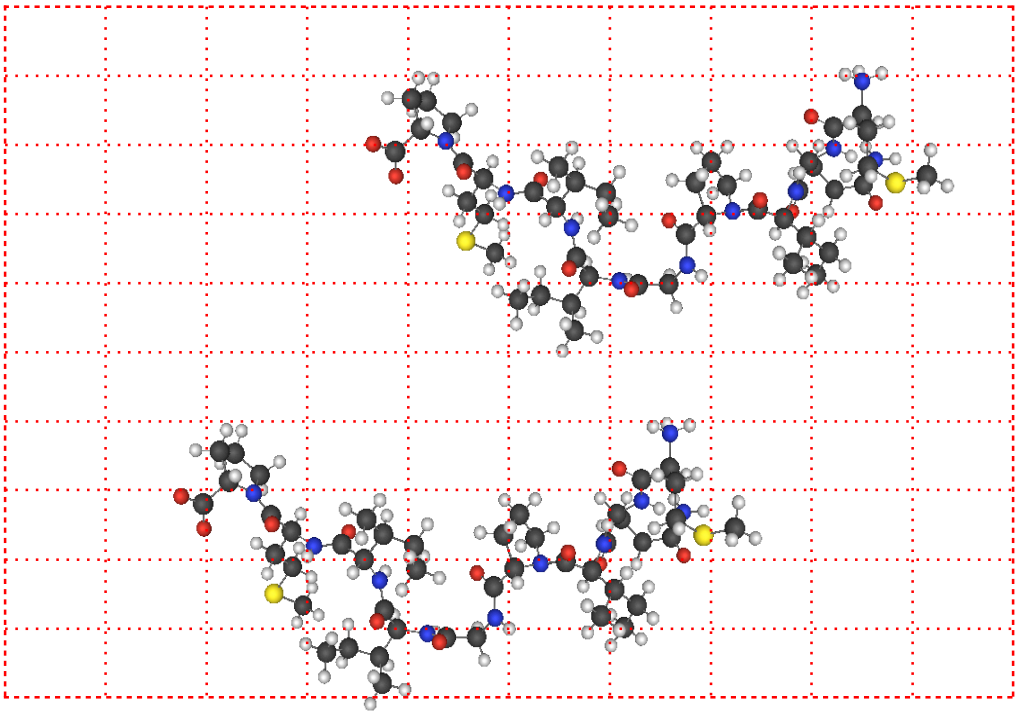
takes relatively a large amount of time for it to access the memory space of a different processor. In some applications, data communication can be less important in that each processor may spend more time processing than sharing data. Molecular dynamics simulations tend not to have this property – data must be communicated quite frequently, and it is this factor that can make parallelization very difficult. One goal in parallelization is thus to minimize the communication of data. Another goal is to keep each processor as busy as possible, that is to minimize the time spent waiting for data to arrive or waiting for data to be synchronized with other processors or with a main memory space.

The two main implementation issues to be addressed are 1) how computation is shared amongst processors, and 2) where and how the data is stored and shared amongst processors.

There are three main approaches to dividing the workload of a molecular dynamics simulations amongst many processors:

1. Divide the force computations amongst the many processors.
2. Divide the atoms amongst the processors.
3. Divide the system spatially, and assign each region of space to a different processor.

The figure below shows how a system consisting of two molecules might be divided spatially amongst many processors. This scheme is hard to load-balance, since some processors might get an area that is very sparsely occupied.



Choosing the most efficient parallelization scheme can depend heavily on the architecture or hardware it is run on. One very successful implementation that almost linearly with the number of processors is NAMD[5], a program which utilizes the Charm++ parallel library[6]. Broadly speaking, NAMD combines both the spatial and force decomposition paradigms to balance the workload very evenly amongst all processors involved. Communication overhead is minimized and is avoided by allowing computation to be performed even while communication takes place.

The second issue is where how where will the complete system be maintained in memory. This includes the positions and velocities of every atom in the system, as well as other atomic properties such as charge and mass. Three schemes may be considered:

1. Keep copies of the system in the local memory of every processor. Every processor computes its share of the load, and then communicates any changes made to other processors, which may also need to use some of the resulting data in the next computation step.

2. Divide the system evenly amongst all the processors, giving each processor all the information necessary for the computations it is responsible for. A small amount of data may need to be duplicated amongst the different processors, depending on how the computation is divided up.

3. Keep the complete system in the local memory of one processor, which then communicates the data necessary for each processor. Each processor does the neccessary processing and communicates the results back to the main processor.

2.1 Parallelizing MD on the Multi-core Architecture

The multi-core chip consists of up to 8 'worker' cores (SPE) that are capable of independently processing data, and a main processing core (PPE) that is responsible for basic operating tasks as well as distributing the work and initiating the processing on each worker core. The main core can access the main memory store, which is similar to the memory on any conventional computer. Each worker core has a local memory store of 256KB, which it can access very efficiently and do computation on. The communication between processors comes through a memory bus which has a circular inter-connect, and can ferry data between main memory and the local store of each SPE core. The program or code that each SPE runs is written in C and loaded into each SPE's local memory. This code includes the necessary communication of data from main memory or from local stores of other SPE cores, as well as the processing algorithm.

Since each SPE core has a rather small amount of local memory that can be directly processed, this means that the first two memory-management schemes in general will not be applicable, and we are restricted to the third memory scheme. Thus the complete system will be stored in the local memory of

[5] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant KalÈ, Klaus Schulten (2005) "Scalable molecular dynamics with NAMD" Journal of Computational Chemistry, vol 26, 16, 1781-1802

[6] L.V. Kale and Sanjeev Krishnan (1993) "CHARM++: A Portable Concurrent Object Oriented System Based On C++" Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications, ACM Sigplan Notes, Vol. 28, No. 10, pp. 91-108

the main processor. The parallelization scheme that will be further explored here is the force decomposition scheme.

2.2 Implementation

The most time-consuming step in a MD simulation is the pair-wise computation of non-bonded interactions, fundamentally of $O(n^2)$ complexity, where n is the number of atoms in the system. To reduce this workload, a distance cut-off is typically employed, so that forces are computed for atoms that are within a certain distance of one another. The distance used varies, however typically it ranges between 1.2nm and 1.8nm. Even with this cut-off, the time for non-bonded force computations still dominates simulation time. Hence, in this work, the focus will be to parallelize these particular computations. Thus a basic force computation involves two atoms, and the result is the force vector for each atom.

The simplified outline of a MD simulation is as follows:

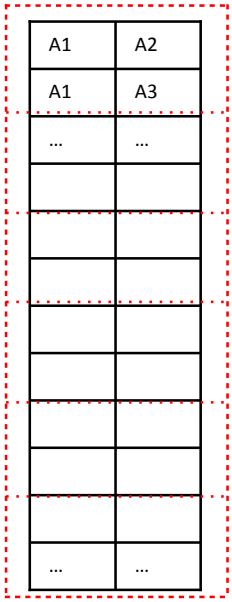
```
For i=0 to numsteps
       if i % ap_freq == 0
              find atom pairs

       Compute 'bonded' forces
              bonds, angles, dihedrals, impropers

       Compute 'non-bonded' forces
              atom pairs

       Integrate Langevin equation for each atom
```

The process of finding atom-pairs that are within the cut-off distance of one another is quite time-consuming itself, but assuming that atoms don't move very far at every step, it can be done every ap_freq steps – a typical value for ap_freq might be 10 for example. An illustration of the pair-wise list is given to the right, with red oulines representing how the computation will be split amongst the available processors. In the figure, A1 refers to information related to atom 1, such as position, charge, and all other parameters related to the non-bonded computation, A2 similarly for atom 2, and so on.



Thus in the parallelization scheme implemented here, this list of pairwise force computations is split evenly amongst the available SPU cores. The MD simulation outline which runs on the main processor (PPU) is as follows:

```
for j=0 to #SPUs
       send initialization control block to SPU-j

for step i=0 to num_steps
```

```
if i % ap_freq == 0
       find atom pairs

compute bonded forces

while non-bonded operations remaining

       for j=0 to #SPUs
              create block with force-operations (200KB)
              send control block with #ops to SPU-j
              tell SPU-j to start processing

       for j=0 to #SPUs
              if SPU-j finished, add forces to atoms
       break if all SPUs finished

       integrate forces
```

The code outline that each SPU core runs is:

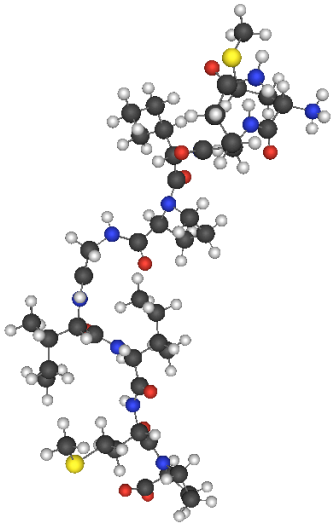
```
receive initialization block with pointer to atom list in main memory
while 1
       receive control block with # force operations and address in main memory
       get atom structures from main memory
       perform force computation
       send resulting forces to main memory
```

As can be seen in the PPU outline, the number of cores utilized can be varied. The main non-bonded force computation loop divides the pairwise atom force computations amongst these cores. For example there might be 1000 pairwise atoms force computations to perform. Since each SPU can only store information for about 200 pairwise atoms at a time, this list is split up into blocks of 200 operations, which are sent sequentially to each SPU. Once an SPU receives the control block, it knows how many operations it has to perform, and the place in main memory where the necessary data is. Thus it first loads this data in its local store, performs the computations, and then results back to a specific address in main memory.

This implementation can be further optimized. Note that each SPU first waits for all data to be received before starting computation. This can be improved by starting computation as soon as the enouh data is received for each force computation.

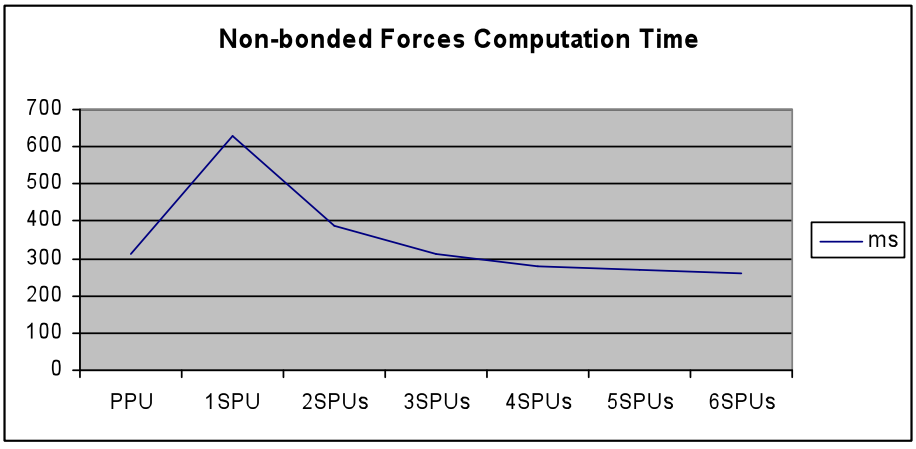
3. Results

The molecule used in the test simulations is a short peptide, which is illustrated below. It consists of 146 atoms in total.



The time for non-bonded computations at each MD simulation step, in ms, as a function of number of processors used is tabulated and graphed below:

PPU	1SPU	2SPUs	3SPUs	4SPUs	5SPUs	6SPUs
310	630	390	310	280	270	260



The data shows that when running the simulation simply on the PPU, the non-bonded force computation at each step takes 310ms. Utilizing one SPU core to do the same computation, the time increases drastically to 630ms. The time increase is simply attributed to the now added communication

time between SPU and main memory. As more SPUs are added, the time per step decreases as the computations are divided amongst more cores and the extra expense in communication time is made up for with parallel computation. With 6 SPUs, the time of 260ms is smaller than the time to run the same computation on the single PPU.

4. Conclusions

A parallel implementation of an MD simulation specifically targeted for the Playstation multi-core chip has been presented. Although the speed-up attained while utilizing 6 cores simultaneously was not formidable, further optimizations may further improve the results.