Extension of the AMBER Molecular Dynamics Software to Intel's Many Integrated Core (MIC) Architecture

Perri J. Needham¹, Ashraf Bhuiyan², Ross C. Walker^{*,1,3}

¹*San Diego Supercomputer Center, University of California San Diego, 9500 Gilman Dr. MC 0505, La Jolla, CA, 92093-0505, United States*

²*Intel Inc., Jones Farm Campus, 2111 NE 25th Avenue Hillsboro, OR 97124, United States*

³*Department of Chemistry and Biochemistry, University of California San Diego, 9500 Gilman Dr. MC 0505, La Jolla, CA, 92093-0505, United States*

E-mail: ross@rosswalker.co.uk

*To whom correspondence should be addressed

Abstract

We present an implementation of explicit solvent particle mesh Ewald (PME) classical molecular dynamics (MD) within the PMEMD molecular dynamics engine, that forms part of the AMBER v14 MD software package, that makes use of Intel Xeon Phi coprocessors by offloading portions of the PME direct summation and neighbor list build to the coprocessor. We refer to this implementation as *pmemd MIC offload* and in this paper present the technical details of the algorithm, including basic models for MPI and OpenMP configuration, and analyze the resultant performance. The algorithm provides the best performance improvement for large systems (>400,000 atoms), achieving a ~35% performance improvement for satellite tobacco mosaic virus (1,067,095 atoms)

when 2 Intel E5-2697v2 processors (2 x 12 cores, 30M cache, 2.7GHz) are coupled to an Intel Xeon Phi coprocessor (Model 7120P – 1.238/1.333 GHz, 61 cores). The implementation utilizes a two-fold decomposition strategy: spatial decomposition using an MPI library and thread-based decomposition using OpenMP. We also present compiler optimization settings that improve the performance on Intel Xeon processors, while retaining simulation accuracy.

Keywords: Xeon Phi; Molecular Dynamics; Code optimization; Acceleration; MIC.

1 Introduction

For many scientific research fields, simulating changes in the structure of large biological molecules, such as proteins, nucleic acids and lipid bilayers, is crucial to understanding important biological pathways and mechanisms [1-12]. Biologically significant structural changes that reveal the mechanisms for the underlying functionality of an enzyme often occur on the microsecond to millisecond timescale [4, 13]. Until recent years such timescales have typically, for most researchers who do not have access to significant computing resources, been out of reach of classical MD simulation for large (> 10,000 atoms) biomolecular systems [13-15]. Key to improving the fidelity of MD simulations is increasing the rate at which conformational space can be explored. This not only makes it possible to study events that occur on longer timescales but also improves convergence of current simulation and provides a mechanism by which to more rapidly validate classical MD force fields. While theoretical innovations such as the annihilation of the fastest vibrational frequencies by the use of constraint algorithms such as SHAKE [16], allowing longer time steps, and the use of advanced sampling methods such as accelerated MD [17], Replica-Exchange MD (REMD) [18], self-guided Langevin dynamics [19] and targeted MD [20, 21] help improve sampling and provide ways to access events that occur on longer timescales it has been advances in computing technology that have had the biggest impact on the typical timescales that can be accessed with classical MD simulations [9, 22-25].

In recent years coprocessor technology has become prevalent in computational science, offering an affordable solution to accelerating computational research. The Amber MD software package[26] is a prime example of the successful use of coprocessor technology. The *pmemd* engine in AMBER is a highly optimized MD simulation package capable of simulating explicitly or implicitly solvated molecular systems using multiple CPUs and/or Graphics Processing units (GPUs) with the AMBER and CHARMM force fields[23, 25]. Running a simulation in parallel on a GPU provides substantial performance increases when compared to multicore CPU simulations with hardware of equivalent monetary value. The latest AMBER 14 *pmemd* benchmark data for an NVE simulation of Cellulose (408,609 atoms) using a 2fs time step performs at 0.85 ns day⁻¹ on 16-cores of 2 x Intel E5-2660v2 compared with 13.36 ns day⁻¹ on 2 x NVIDIA GTX Titan Black GPUs[27]. To put this into context, using the aforementioned GPU setup would enable a microsecond simulation of Cellulose in 75 days as opposed to 1177 days with a CPU setup of roughly equivalent monetary value.

A more recent addition to the coprocessor market is Intel's Xeon Phi Many-Integrated core (MIC) architecture. At the time of writing, this family of coprocessors offers up to 61 x Intel x86 cores running at 1.238 GHz (1.333 GHz turbo clock) with 16GB of onboard memory providing up to 1.2 teraFLOPS of double precision floating-point performance accessible via a PCI Express expansion card. The cores are arranged in a bidirectional ring network (see figure 1) and each have their own 512 KB L2 cache offering full cache coherency across all cores. Hyper-threading technology supports up to 4 threads per core and each thread can execute two instructions per cycle. This, in principal, allows latency to be hidden by allowing non-executing threads to prefetch data from memory. One vector instruction can be issued to the 512-bit SIMD vector units per cycle and fused multiply-add is supported.

Figure 1: Intel Many-Integrated Core (MIC) architecture featuring a bidirectional ring interconnect connecting cores, L2 caches, tag directories (TD), the PCIe Client Logic and the GDDR5 memory controllers.

This paper describes the introduction of Xeon Phi support, on the Knight's Corner generation of Intel's architecture, in Amber's *pmemd* MD engine for improving the

performance of explicit solvent (particle mesh Ewald – PME) MD simulations. While implicit solvent Generalized-Born MD simulations are supported, they are not described in this manuscript. Previous published attempts at optimizing MD using Intel's Xeon Phi architecture have relied on heavy code modification, SIMD vectorization and Intel's Thread Building Blocks (TBB)[28-31]. The approach taken in this work avoids extensive code vectorization and tuning specific to the Xeon Phi core as only a marginal performance improvement (< 5%) was achieved by using vector instructions and the code could require extensive rewriting for each new generation of Xeon Phi accelerator. The limited performance improvement seen from extensive explicit vectorization of the existing *pmemd* code is due to irregular memory access patterns within loops and addressing this comes at the expense of code maintainability and readability. This implementation also avoids large algorithmic changes and instead focuses on maximizing flexibility while also achieving performance improvements. The approach taken was to utilize the existing Message Passing Interface (MPI) framework to offload portions of the direct space calculation to the Xeon Phi coprocessor and then further decompose each offloading MPI task using OpenMP threading to expose fine-grain parallelism whilst minimizing the memory footprint and optimizing memory reuse.

This technical paper describes the design of the *offloading* algorithm and the performance of *pmemd* when simulating large biomolecular systems in *offload mode*. Section 2 of this paper gives an overview of the MD theory relevant to this work and the Intel Xeon Phi coprocessor programming modes that are available. Section 3 covers the technical details of the *pmemd offload* algorithm including the decomposition strategy employed. Section 4 presents the performance of the *pmemd offload* implementation for large biomolecular systems and section 5 provides initial conclusions and a discussion of future work.

2 Theory

2.1 Overview of MD

Molecular dynamics (MD) is a means of simulating molecular systems in order to predict the physical motion of their composite atoms/particles. Monitoring the evolution of a molecular system exposes significant physical and chemical behaviors, revealing important biological phenomena such as protein folding, drug binding and biological catalysis, that would otherwise be difficult, if not impossible, to observe experimentally. For this reason molecular dynamics is used extensively as a tool in the drug discovery process by the pharmaceutical and biotech industries[13, 32]. Other research areas make use of molecular dynamics in biocatalysis, enzyme engineering and advanced biomaterials design. MD of biological systems, in its most widely used form, utilizes classical mechanics and a set of pre-determined molecular mechanics force fields, which have been parameterized from more accurate quantum mechanical data or fit to experimental observables, to evaluate the gradient of the potential energy of a given set of atomic coordinates as a function of time. This enables Newton's equations of motion, with a suitable integrator, to be used to propagate atomic positions through time. An MD time step invariably refers to the evaluation of the molecular system at intervals on the scale of atomic fluctuations (typically 2fs intervals). Such short time steps are typically needed to capture the fastest vibrational motions but at the same time it is desirable to simulate microseconds or longer. This implies linear simulations requiring on the order of 10^9 integration steps.

In a typical explicit solvent simulation the interaction of all the atoms present in the solvent (water + any additional ions) are discretely calculated along with the solute. In Amber the potential energy of a given configuration of atoms is, formally, described by the following equation[33]:

$$\begin{aligned}
V_{\text{AMBER}} = & \sum_i^{n_{\text{bonds}}} b_i (r_i - r_{i,eq})^2 \\
& + \sum_i^{n_{\text{angles}}} a_i (\theta_i - \theta_{i,eq})^2 \\
& + \sum_i^{n_{\text{dihedral}}} \sum_n^{n_{i,\text{max}}} \left(\frac{V_{i,n}}{2} \right)^2 [1 + \cos(n\phi_i - \gamma_{i,n})] \\
& + \sum_{i < j}^{n_{\text{atoms}}} \left(\frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^6} \right) + \sum_{i < j}^{n_{\text{atoms}}} \left(\frac{q_i q_j}{4\pi\epsilon_0 r_{ij}} \right)
\end{aligned} \tag{1}$$

where the summations involving bonds and angles are calculated from harmonic functions using force constants b_i and a_i and equilibrium bond distances $r_{i,eq}$ and angles $\theta_{i,eq}$. The third component in equation 1 represents the dihedral energy and uses a truncated Fourier series comprising a torsional constant $V_{i,n}$ for each individual Fourier term n and phase shift $\gamma_{i,n}$. n typically runs from 1 to 4. These are the bonded terms in the potential energy function and are dependent on the covalent bonding in the structure being simulated. The non-bonded terms, the last two terms in equation 1, describe the interaction of an atom with all other atoms in the system separated by more than 3 bonds (denoted by a prime). These are the van der Waals interactions, which are represented by a 12-6 Lennard-Jones potential using diatomic parameters A_{ij} and B_{ij} , and the Coulomb interactions calculated from atomic point charges q_i and q_j and the distance separating them r_{ij} . Periodic boundary conditions are normally used in explicit solvent simulations such that particles leaving one edge of the simulation box re-enter on the opposite side. This can be expressed mathematically as:

$$V_{\text{vdW}} = \sum_{\mathbf{n}} \sum_{i < j}^{n_{\text{atoms}}} \left(\frac{A_{ij}}{r_{ij,\mathbf{n}}^{12}} - \frac{B_{ij}}{r_{ij,\mathbf{n}}^6} \right) \tag{2}$$

$$V_{\text{Coulomb}} = \sum_{\mathbf{n}} \sum_{i < j}^{n_{\text{atoms}}} \left(\frac{q_i q_j}{4\pi\epsilon_0 r_{ij,\mathbf{n}}} \right) = V_{\text{direct}} + V_{\text{reciprocal}} + V_{\text{self}} \quad (3)$$

where, $\mathbf{n}$ indexes the periodic copy of the system. Naively computing the non-bonded interactions is an $O(n^2)$ problem and therefore, for large periodic systems the computational bottleneck. As the vdW interactions become negligible with increasing distance between particles a cutoff typically on the order of 8 to 12 Å, plus a correction factor, can be applied to reduce the computational workload. Coulomb interactions, however, decay as $1/r$ and thus have a much broader reach. This means that application of a simple cutoff can lead to truncation of the interactions and as a consequence, non-conservation of energy and unphysical chemical behavior as the simulation progresses with time. To reduce the computational workload of the Coulomb interactions from what is formally $O(n^2)$ to a more manageable $O(n \log(n))$, *pmemd* employs the particle mesh Ewald summation[34]. This approximation partitions the problem into three fast converging components: the direct sum (eq. 4), the reciprocal sum (eq. 5) and a self-correction term (eq. 6).

$$V_{\text{direct}} = \frac{1}{2} \sum_{\mathbf{n}} \sum_i^{n_{\text{atoms}}} q_i q_j \frac{\text{erfc}(\alpha r_{ij,\mathbf{n}})}{r_{ij,\mathbf{n}}} \quad (4)$$

All pairwise interactions that fall within the cutoff radius are computed explicitly as part of the direct sum. The direct sum replaces the summation of interparticle point charges with a sum of point charges q_i and q_j on particles i and j screened by Gaussian distributions to ensure the strength of the interaction falls off rapidly with increasing distance $r_{ij,\mathbf{n}}$. This enables the use of a small direct space cutoff (≈ 8 -12 Å). The Gaussian charge distributions are centered on the same positions as the point charges with the same magnitude but opposite sign. They are represented by the complementary error function $\text{erfc}(x)$ where x involves an Ewald parameter α for screening the charge. This approach introduces additional charge density, which is dealt with by summing Gaussian charge distributions placed at the same positions as the point charges with the same

magnitude and this time also the same sign. These periodic charges are distributed onto a grid and solved using the Fourier series by conversion into reciprocal space, hence the term the reciprocal sum.

$$V_{\text{reciprocal}} = \frac{1}{2\pi v} \sum_{\mathbf{m} \neq 0} \frac{\exp\left(-\left(\frac{\pi \mathbf{m}}{\alpha}\right)^2\right)}{\mathbf{m}^2} F(Q)(\mathbf{m}) F(Q)(-\mathbf{m}) \quad (5)$$

where v is the volume of the unit cell, $\mathbf{m}$ is a reciprocal lattice vector and $F(Q)$ is the fast Fourier transform of Q , allowing for faster convergence as the problem is reduced to $O(n \log(n))$. Q is a charge matrix that represents the charges interpolated onto a 3D grid representation of the simulation box.

The final component of the PME summation is the self-correction, which removes interactions of an atom with itself that arise from the additional charges introduced by the reciprocal term.

$$V_{\text{self}} = \frac{-\alpha}{\sqrt{\pi}} \sum_{i=1}^{n_{\text{atoms}}} q_i^2 \quad (6)$$

The three PME terms are analytical differentiated to give the Coulombic forces.

2.2 Intel Xeon Phi Programming Modes

The Knights Corner version of the Intel Xeon Phi coprocessor has 3 different modes of programming: *symmetric*, *native* and *offload* mode. The Molecular Dynamics (MD) implementation that uses the Particle-Mesh Ewald (PME) summation to optimize the calculation of long-range electrostatic interactions in the Amber MD package, namely *PMEMD*, was extended as part of this work to support the use of Intel Xeon Phi coprocessors in *native* and *offload* mode. *Symmetric mode* is not used in this work and thus is not discussed further. In the remainder of this paper the terms Many Integrated core (MIC) card and device are used interchangeably, and refer to the Intel Xeon Phi coprocessor. The terms CPU and host are also used interchangeably, and refer to the CPU that the MIC is coupled to.

One advantage of using Intel's Xeon Phi coprocessors is the ease by which pre-existing parallel code can be executed on the architecture. By recompiling with MIC flags, MPI tasks/OpenMP threads can execute in *MIC native mode* across the MIC cores without modification of the code. The entire application runs on the device, moving away from the traditional coprocessor paradigm to one that treats the hardware more like an integrated compute node. However, it is the responsibility of the user/developer to transfer the relevant files, such as input and binary files, to the device's memory and a performance improvement is not guaranteed as the pre-existing parallel program may not have been optimized for execution on MIC cores.

MIC offload mode offers a more traditional coprocessor paradigm, giving control to the developer to make use of the MIC cores by offloading portions of work to the device. However, it involves time and effort to modify the code and a degree of many-core programming knowledge to fully exploit the MIC resources. The MIC hardware supports Intel AVX and SSE vector instructions for making use of 512-bit SIMD vector units coupled to each processor core and it is often necessary to program specifically to take advantage of these vector units in order to achieve acceptable performance.

3 Technical Details of the Implementation

Presented in this section are details of a PME MD algorithm that has been developed to offload portions of the calculation to an Intel Xeon Phi coprocessor as a means of accelerating the simulation. Figure 2 illustrates the basic execution path taken when computing the equations given above for a PME MD simulation using the *pmemd* engine in AMBER. A more in-depth description of the offloading decomposition strategy for each energy contribution in the calculation above (eq. 1), along with the neighbor list algorithm (giving special attention to the PME direct space sum and the building of the neighbor list, as these are the focus of acceleration) follows later in this section. Additional features and functionality, such as constraints, restraints, thermostats and barostats, have been omitted to simplify the workflow diagram. Program execution begins with simulation setup, which involves inputting the atomic coordinates,

parameters and other force field data from the system input files. After the initial simulation setup the algorithm checks if the neighbor list needs to be created/updated. On the first time step, as the neighbor list has not been built previously, this check will result in the creation of the neighbor list. On subsequent time steps the neighbor list is rebuilt if the movement of atoms on the previous time step has resulted in new neighboring atoms. If no updates to the neighbor list are required then execution of the neighbor list build routine is skipped for the current time step. An explanation of the algorithm used to build the neighbor list is provided below when describing the approach used to offload portions of the workload to an Intel MIC coprocessor. The force calculation is computed in 4 separate routines: the PME self-correction (eq. 6); the PME direct space sum (eq. 4) and the vdW calculation (eq. 2), which are packaged together for computational efficiency as both rely on the use of a cutoff; the PME reciprocal space sum (eq. 5); and the calculation of the covalently bonded (bond, angle and torsional) force contributions (the first three terms in eq. 1). These separate force contributions are summed together to give the overall forces on the atoms allowing the velocities and atomic positions to be updated using Newton's equations of motion. If the number of time steps has not yet reached the maximum then the neighbor list and forces are computed again with the updated coordinates at $t = t + \delta t$ where δt represents the time increment, which is commonly set to 2fs. Once the specified number of time steps has been computed the simulation is stopped.

The approach used, in this work, to offload computation to the Intel MIC coprocessor was to further optimize the pre-existing MPI implementation by accelerating computationally intensive calculations/routines using OpenMP threads. The decomposition strategy employed has a dualistic-nature as the molecular system is dynamically decomposed spatially across MPI tasks in a coarse-grained manner and then more finely decomposed across OpenMP threads on an Intel MIC coprocessor in a static thread-based fashion.

Figure 2: Workflow diagram showing the execution path of *pmemd* on each time step.

1
2
3
4 In order to determine the most effective use of an Intel Xeon Phi coprocessor the
5 execution of an MD simulation of a molecule of cellulose (408,609 atoms) in the
6 isothermal-isobaric (NPT) ensemble using 24 MPI tasks distributed across 24 cores (2 x
7 Intel Xeon E5-2697 v2) was analyzed. Figure 3 shows the proportion of time spent by all
8 MPI tasks in executing: the PME neighbor list; the PME direct space contribution; the
9 PME reciprocal space contribution; the PME bonded force contribution; data
10 distributions; and other routines/calculations. On analysis it can be seen that a natural
11 choice to focus acceleration efforts on is the PME direct space sum, as this would have
12 the highest impact on performance due to being the most computationally intense
13 calculation, taking over 44% of the program's execution time. The calculation of the
14 PME direct space sum relies on the neighbor list to identify atom pairs that fall within
15 the designated cutoff distance. This dependency of the PME direct space calculation on
16 the neighbor list motivated the decision to also execute part of the neighbor list building
17 routine on the MIC. The decision came after investigation of the alternative, which
18 involved computationally expensive data transfers between the host and the MIC to
19 ensure the PME direct space routine could access an up-to-date neighbor list, proved to
20 be detrimental to performance. Further motivation for accelerating both the PME
21 direct space calculation and the neighbor list building routine came from the two
22 routines using the same load-balancing scheme to dynamically assign atoms to MPI
23 tasks, which greatly simplified the programming effort.

24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43 **Figure 3: Chart to show the distribution of work between the main components of the PME force calculation in**
44 **AMBER which are: the PME neighbor list build; the PME direct force calculation; the PME reciprocal force**
45 **calculation; the PME bonded force calculation; the time taken to distribute data; and the other remaining**
46 **components.**
47
48
49

50 **3.1 Offloading MPI tasks**

51
52
53 The approach used to offload portions of work to the Intel Xeon Phi coprocessor was to
54 develop a 2-fold decomposition strategy combining spatial decomposition with thread-
55 based decomposition. The spatial decomposition of the atoms in a system across MPI
56 tasks has been optimized to minimize communication between tasks. The workload of a
57
58
59
60
61
62
63
64
65

subset of the executing MPI tasks is offloaded to the Intel Xeon Phi coprocessor and is further decomposed using a more fine-grained thread decomposition across the many-integrated cores (MIC). The number of offloading MPI tasks is determined at system setup via the following pseudo-code,

```
if the system size is less than or equal to 100,000 atoms then
    No. of offloading MPI tasks = 2
else
    No. of offloading MPI tasks = 4
end if

if the total number of MPI tasks is greater than 12 and less than 16 then
    No. of offloading MPI tasks = 2
else if the total number of MPI tasks is less than 12
    No. of offloading MPI tasks = 1
end if
```

The number of offloading MPI tasks is conditional upon two variables; the number of MPI tasks launched and the number of atoms in the system. For systems of 100,000 atoms and above 4 MPI tasks offload work to the MIC otherwise only 2 MPI tasks offload to the MIC due to the overhead of each MPI task becoming more significant at smaller system sizes. However, the number of offloading MPI tasks is adjusted if less than 16 MPI tasks are launched in total. If the total number of MPI tasks launched is between 12 and 16 then 2 MPI tasks offload to the MIC while if the total number equals 12 or less only 1 MPI task offloads to the MIC. This ensures enough MPI tasks are available to parallelize the CPU workload. The results of an investigation into the optimal number of offloading MPI tasks for a simulation of cellulose (408,609 atoms) can be seen in figure 4. The performance of the simulation is optimal when 4 MPI tasks are offloading to the MIC. Fewer than 4 MPI tasks means more OpenMP threads have to be launched per MPI task to prevent idle cores. This has the effect of increasing the size of private arrays on the MIC, causing more memory (cache and page) misses. More

than 4 MPI tasks offloading also results in suboptimal performance, as the amount of computation is insufficient to amortize the start-up costs and overheads involved for each offloading MPI task.

Figure 4: Performance in ns day⁻¹ when varying the number of MPI tasks that offload work to an Intel Xeon Phi coprocessor. The platform used was 2 Intel IVB E5 2697v2 (2 x 12 cores) + 1 7120P Xeon Phi (61 cores) using 24 MPI tasks.

Based on the findings above a basic model has been developed for determining the number of MPI tasks on a CPU and a MIC:

- 1) Total number of MPI tasks on the host/CPU:

The number of MPI tasks = the number of host/CPU cores.

- 2) Number of offloading MPI tasks:

Since there are two DMA channels between a CPU and a MIC, generally the number of offloading MPI tasks = 2.

Exception 1) 2 MPI tasks < 20% total MPI tasks. In this case the number of offloading MPI tasks = 4.

Exception 2) 2 MPI tasks constitute < 20% total workload. In this case the number of offloading MPI tasks = 4.

The example in figure 5 illustrates the 2-fold decomposition strategy and the offloading scheme for a system size of less than 100,000 atoms with 9 MPI tasks launched at runtime. In this scenario, only 1 MPI task offloads to the MIC. The offloading MPI task spawns OMP threads and decomposes the workload/atoms across the OMP threads in as close to equal portions as possible. The offloading scheme selects the range of MPI rank(s) to assign work to offload to the MIC coprocessor via the following pseudo-code,

if the system size is less than or equal to 100,000 atoms then

first offloading MPI rank = (total number of MPI tasks)/2-1

last offloading MPI rank = (total number of MPI tasks)/2

else

the first offloading MPI rank = (total number of MPI tasks)/2-2

the last offloading MPI rank = (total number of MPI tasks)/2+1

```

end if

if the total number of MPI tasks is greater than 12 and less than 16 then
    of the first offloading MPI rank = (total number of MPI tasks)/2-1
    the last offloading MPI rank = (total number of MPI tasks)/2
else if the total number of MPI tasks is less than 12
    the first offloading MPI rank = (total number of MPI tasks)/2
    the last offloading MPI rank = (total number of MPI tasks)/2
end if

```

For example, in figure 5 each square represents the atoms to be processed by a single MPI task. The MPI tasks are ranked from 1 to 9 and follow row-major order. The offloading scheme assigns MPI rank 5 (represented by the shaded centrally located square) as the offloading MPI task. This MPI task will offload a portion of its work to the MIC coprocessor for execution. The offloading MPI rank(s) is(are) selected in a manner that exploits data locality within the periodic boundary condition framework to optimize the performance of transferring data, such as the coordinates and the forces, to and from the MIC.

Figure 5: An example to demonstrate the 2-fold decomposition scheme of the MIC *offload* algorithm in AMBER. On the left: coarse-grained spacial decomposition of a system of atoms on a CPU using MPI with each box of atoms being processed by one MPI task. On the right: a single MPI offloads work to the Many-Integrated Core (MIC) architecture, spawns OpenMP threads, which execute evenly distributed portions of the workload in a fine-grained thread-based decomposition.

3.2 OpenMP threads on the MIC

To decompose the workload on the MIC it is first partitioned into as many parts as there are offloading MPI tasks to promote data locality, as shown in figure 6. For instance, the MIC cores are partitioned in to two groups if 2 MPI tasks are offloading and four groups if 4 MPI tasks are offloading. Each physical MIC-core is capable of launching 4 OMP threads so for an Intel Xeon Phi coprocessor made up of 60 cores a total of 240 threads can be launched. However, investigation has shown that it is not always beneficial to launch the maximum number of threads as this can deplete memory on the MIC so for

1
2
3
4 this reason, less OMP threads are launched for systems of greater than 0.5 million
5 atoms. For instance, 4 offloading MPI tasks would each launch 40 OMP threads (160
6 OMP threads in total) on the MIC instead of 60 when simulating a 1 million-atom
7 system. The distribution scheme used to map the OMP threads to physical cores in each
8 partition is dependent on the number of OMP threads launched per core. If less than
9 the maximum number of OMP threads per core is launched (<4 OMP threads) then a
10 scattered distribution scheme is used to ensure load balancing by evenly distributing
11 threads to cores. If the maximum number of OMP threads per core is launched (4 OMP
12 threads) then a balanced distribution scheme is employed, as load balancing is not
13 required and it improves the readability of the code.
14
15
16
17
18
19
20
21
22
23

24 **Figure 6: Demonstration of the offloading scheme in the *pmemd* MIC offload algorithm. HOST: 16 MPI tasks are**
25 **launched with the 2 middle MPI ranks (MPIs 8 & 9) offloading 78% of their workload to the MIC; MIC: 60 cores of a**
26 **MIC coprocessor are partitioned into two parts, one for each offloading MPI task. The offloading MPI tasks are**
27 **capable of spawning up to 4 OpenMP threads per core in their partition ranked from core 1-120.**
28
29

30 A general model for choosing the number of OpenMP threads to be spawned on a MIC
31 would be,
32
33

$$\text{No. of total threads} = \text{No. of cores on MIC} * 4$$

34 Building on this OpenMP and the aforementioned MPI model, better performance can
35 potentially be achieved by tuning the number of MPI tasks and OpenMP threads to suite
36 a particular molecularsystem, CPU/MIC architecture or CPU/MIC arrangement in a
37 pragmatic fashion.
38
39
40
41
42
43
44

45 **3.3 Load-balancing scheme**

46
47 The work involved in calculating the forces on the atoms as described by the equations
48 above is distributed to the MPI tasks as follows:
49
50

- 51 • PME direct sum and neighbor list – all MPI tasks are assigned PME direct sum and
52 neighbor list work. The load-balancer monitors the performance of each MPI task in
53 calculating the PME direct sum over steadily increasing time step windows and
54 dynamically adjusts the amount of PME direct sum and neighbor list work assigned
55 to each MPI task accordingly. The neighbor list performance for each MPI task is not
56
57
58
59
60
61
62
63
64
65

monitored as it is directly proportional to the performance of the MPI task in calculating the PME direct sum and so is automatically adjusted based on this. Figure 7 illustrates the dynamic load-balancing scheme employed in the *pmemd* MIC offload algorithm.

Figure 7: Dynamic load balancing of the PME direct force and neighbor list workload as the simulation progresses. Top left: the workload is evenly distributed amongst threads at 0 time steps (ts). Top right: after 15 time steps the load-balancer gives more work to the MIC offloading MPI task. Bottom left: after 30 time steps the MIC offloading task has been given an even larger portion of the workload. Bottom right: after 45 time steps the MIC offloading task has roughly 50% of the total workload. N.B. The workload distribution in this diagram is given in the abstract form in order to illustrate the load-balancing scheme.

At the start of a simulation offloading MPI tasks outperform non-offloading MPI tasks and so, as the simulation progresses, a larger proportion of the work is assigned to the offloading MPI tasks by the dynamic load balancer. Offloading MPI tasks execute 78% of the PME direct space sum and neighbor list work on the MIC and the remainder of the workload is executed on the CPU concurrently as shown in figure 6. A CPU:MIC workload ratio of 22:78 achieved the most optimal performance when simulating a system of cellulose and was deduced by testing the performance at different ratios, as can be seen in figure 8. However, this ratio is system specific and so a crude model to determine the CPU:MIC workload ratio for other molecular systems or hardware arrangements has been developed and is provided in subsection 3.3.1 below.

It is important to note that, although the CPU:MIC workload distribution ratio is static the actual amount of work offloaded to the MIC by the offloading MPI task(s) is nonetheless still dynamically adjusted. This is because the MPI task load-balancer monitors the performance of each MPI task and assigns work based on this metric. The offloading MPI tasks complete their workload at a different pace to the non-offloading MPI tasks and so workload is redistributed accordingly. This means that the actual amount of work executed on the MIC is adjusted as the workload is redistributed across all MPI tasks throughout the course of a simulation. It is

intended for future versions of the MIC *offload* implementation to dynamically adjust the ratio as the simulation progresses through time.

Figure 8: Performance of a cellulose simulation when varying the fraction of work offloaded to the MIC by each offloading MPI task. Hardware: 2 Intel E5-2697v2 processors (2 x 12 cores) + Intel Xeon Phi 7120P coprocessor (61 cores). MPI configuration: 24 MPI tasks launched on the CPU, 4 of which offload work to the MIC.

- PME reciprocal sum work – depending on the number of MPI tasks launched either all or a fraction of the MPI tasks are given atoms for which to calculate the PME reciprocal force contribution according to the model of decisions in Figure 9. For example, a system of Cellulose (408,609 atoms) distributes PME reciprocal work to all MPI tasks; a smaller system would only distribute work to a fraction of the MPI tasks launched. However, as the PME reciprocal sum is not offloaded to the MIC, offloading MPI tasks are restricted from being assigned PME reciprocal work. This is done to promote concurrent execution of the CPU and MIC as, if offloading MPI tasks were assigned PME reciprocal work, execution of offloaded work on the MIC would not be able to proceed until the PME reciprocal work had been executed on the CPU, leaving the MIC idle. However, in the scenario where 10 or less MPI tasks are launched, all get assigned PME reciprocal work. This is done as a safeguard. The load balancer takes into consideration the total number of ranks, the system size, direct force assignment and the relative speed of the processing cores when assigning reciprocal work to MPI tasks. We have introduced a new component *“offloading MPI task not doing reciprocal work”* in the already complicated load-balancing algorithm. Thus, a corner case can arise where only one MPI task will get all the reciprocal work, or even worse, no MPI tasks get reciprocal work. This situation can arise for low numbers of MPI tasks, such as 6 or lower, so a safety bar of 10 MPI tasks has been set for reciprocal work assignment to offloading MPI tasks. The workload is then distributed evenly to the assigned MPI task(s).

Figure 9: Basic decision tree for the assignment of reciprocal work to MPI tasks.

- Bonded force work – a fraction of the MPI tasks launched are assigned bonded force work and the amount of work is dynamically assigned in a similar manner to the PME direct sum and neighbor list work dependent on performance in previous time steps. As is the case with the PME reciprocal sum, bonded work is restricted to non-offloading MPI tasks only as, bonded force contributions are not calculated on the MIC and so asynchronous execution with the MIC can only occur when bonded force contributions are calculated using non-offloading MPI tasks. Again, similar to the PME reciprocal sum, if 10 MPI tasks or fewer are launched then bonded force work is assigned to offloading MPI tasks.

3.3.1 A basic model for determining the CPU:MIC workload distribution ratio for offloading MPI tasks

The *performance* ratio between a CPU and a MIC can, at it's most basic, be approximated by simply considering the number of cores and the clock frequency. This ignores factors that add more complexity such as differences in memory bandwidth, cache size, vector units etc., however, for our simple model core count and clock speed suffice.

If we take a system with 2 x E5 2697 v2 CPUs and a single 7120P MIC card then for the CPU we have a core count of 24 and a clock speed of 2.7GHz. Multiplying these gives a raw performance value of 64.8 (24x2.7) for that particular CPU. Similarly for the 7120P MIC card we get a raw value of 66.0 (60x1.1). We then ratio these two numbers to get a *performance* ratio of 1.00 : 1.01.

According to this simple model, *workload* can be divided almost equally between 2 x E5 2697 v2 CPUs and a 7120P MIC, however, in practice offloading *workload* to a MIC carries an overhead, such as transferring data between a CPU and the MIC. Additionally, some portions of the code offloaded to the MIC may be executed in serial, which as each core of a MIC runs ~2.5x slower than an E5 2697 v2 CPU core, will also detract from performance. The overhead also depends on the type of workload. This leads us to a general ratio for *workload* division between 2 x E5 2697 v2 CPUs and a 7120P MIC of

1: x , where x is less than 1. In many practical cases x varies from 0.3 to 0.7 for different CPU models. For simplicity in generalizing things we can assume x to be the mean value,

$$x = \frac{0.3 + 0.7}{2} = 0.5$$

Therefore, a basic ratio for the division of *work* between a CPU and a MIC can be 1:0.5, which would then need to be adjusted by the raw core x clock frequency ratio depending on the CPU and MIC hardware.

In the scenario where 4 MPI tasks offload to the MIC and 24 MPI tasks are launched across the 24 xE5 2697 v2 cores, the CPU can potentially execute 6x the *workload* of the 4 offloading MPI tasks on the host,

$$\frac{\# \text{ of CPU cores}}{\# \text{ of offloading MPI ranks}} = \frac{24}{4} = 6$$

Therefore, the proportion of workload that can be offloaded to the MIC by the offloading MPI tasks, whilst concurrently executing on the CPU, is $(6 \times 0.5) \approx 3 \times$ the CPU workload of the 4 offloading MPI tasks. This gives an approximate CPU:MIC workload ratio of 1:3 for the offloading MPI tasks. If the MIC can run fraction y of the total workload for the offloading MPI tasks (CPU + MIC) we get the ratio,

$$CPU:MIC = 1 - y : y = 3$$

which gives, $y = 0.75$ and a final workload ratio of,

$$CPU:MIC = 0.25:0.75$$

However, this model is very basic as several approximations have been made and so it should serve as a general guideline and starting point for investigating the best CPU:MIC workload ratio for offloading MPI tasks for a specific hardware setup.

3.4 Neighbor list algorithm

The neighbor list is executed across all MPI tasks and offloading MPI tasks concurrently execute the neighbor list on a CPU and a MIC for their assigned atoms at a workload distribution ratio of 22:78, as mentioned above. The algorithm works by dividing the simulation space into buckets. The width of a bucket is roughly half the cutoff distance. This greatly reduces the computational effort in searching for neighboring atoms, as a

fewer number of neighboring buckets need to be found instead. The steps to build the neighbor list are:

- 1) Offloading MPI tasks find their neighboring buckets on the host.
- 2) The variables that are required to build the neighbor list for offloaded atoms are transferred to the MIC asynchronously.
- 3) Two separate neighbor lists are constructed concurrently:
 - a. Host neighbor list
 - b. MIC neighbor list.

These lists are private to the threads residing on the hardware that the list is built on and are never combined.

- 4) Variables required by the host, such as the variables that hold a record of used atoms *img_used_lst* are asynchronously transferred from the MIC to the host.
- 5) Host execution waits for the return signal from the MIC before continuing.

The MIC neighbor list is statically allocated giving each OpenMP thread a fixed number of memory locations for which to store neighboring atoms. The neighbor list is then reallocated if an atom's number of neighboring atoms grows beyond the memory allocated. This static memory allocation scheme keeps memory usage to a minimum (as only neighboring atoms whose PME direct force is to be computed on the MIC are stored) and reduces the amount of memory allocations (as the neighbor list for each atom might not grow each time it is built).

3.5 PME Direct force algorithm

The PME direct sum is executed on the CPU and the MIC concurrently. The offloading MPI tasks execute the following steps:

- 1) MIC variables are declared and global variables are copied to device memory locations.
- 2) Variables that require updating every time the neighbor list is built are copied or reassigned to the MIC, as the MIC cannot use data in host memory.
- 3) Variables that are required on every time step are asynchronously transferred to the MIC allowing simultaneous execution on the host and the MIC.
- 4) Indexing code allows for identifying neighbors in the statically allocated MIC neighbor list.
- 5) The force calculations start on the MIC using OpenMP threads spawned from offloading MPI tasks.
- 6) Control is immediately passed back to the host allowing asynchronous computation of the PME direct force on the host and MIC.
- 7) Each OpenMP thread has an individual copy of the array used to store the forces, as otherwise race conditions would occur. The use of locks to save memory was investigated and the results showed that, even with the costly initialization and reduction of the multiple copies of the force array when using the private array approach, private arrays performed roughly 10-15% faster than the alternative lock approach. The trade-off is increased memory usage.
- 8) The PME direct force calculation on the host uses a look-up table (LUT) to calculate the complementary error function (erfc in eq. 4). However, on the MIC the LUT created high memory access latency as a consequence of L2 cache misses. These occur frequently because the algorithm involves irregular memory access patterns, which cause poor performance when the MIC attempts to vectorize accesses. By calculating erfc on the fly some of the latency is reduced (however not all as some other arrays i.e. the force

and coordinate arrays, still hold latency overheads due to irregular array accesses) which improves performance by a further 5%.

9) Reduce the force arrays into a single force array once all OpenMP threads have finished.

10) Host execution waits for the return signal from the MIC before continuing.

11) The MIC force array is asynchronously copied back to the host and combined with the host force array.

4 Performance

The performance of the *pmemd* MIC offload algorithm was tested using a series of benchmarks designed to represent typical molecular systems that would be simulated. The benchmark systems are: factor IX (90,906 atoms) – a serine protease of the coagulation system; cellulose (408,609) – a polysaccharide that provides the structure to plant cell walls; and satellite tobacco mosaic virus (stmv) (1,067,095 atoms) – a plant virus that worsens the symptoms of infection caused by the tobacco mosaic virus. Benchmarking was carried out on 2 Intel Xeon E52697 v2 CPUs, referred to here as Ivy Bridge (IVB) (2 x 12 cores, 24 threads, 30M Cache, 2.70 GHz) and 2 Intel Xeon E5 2697 v3 CPUs, referred to here as Haswell (HSW) (2 x 14 cores, 28 threads, 35 M cache, 2.6 GHz) coupled to an Intel Xeon Phi 7120P coprocessor (61 cores, 240 threads, 512 KB cache, 1.1 GHz), also referred to here as MIC. The Intel compiler version 14 update 1[35], Intel MPI version 4.1[36], MPSS version 3.2.3[37] and Flash version 2.1.02.0390 were used. For comparison, performance results when running the PME MD simulations using the equivalent precision model (Double Precision, Fixed Precision – DPFP, 64-bit precision) on a NVIDIA K40 GPU (2880 cores, 745 MHz) are given. It should be noted that this is not the fastest GPU precision model available for PME MD simulations in Amber however this provides the equivalent accuracy to the MIC offload model (64-bit precision) that has been developed. A mixed precision model (Single Precision, Double Precision - SPDP) of the MIC offload implementation is currently under development. All

simulations were carried out with ECC on and overclocking features were not used on any hardware.

The optimization flags used for FORTRAN source code are: `-align array64byte -ip -O3 -no-prec-div -xHost -openmp -opt-streaming-cache-evict=0 -fimf-domain-exclusion=15 -fp-model fast=2`; and for C/C++ source code are: `-ip -O3 -no-prec-div -xHost -openmp -opt-streaming-cache-evict=0`. The compiler flags used here that are different from those that have historically been used for AMBER when building with the Intel compilers are:

- `-align array64byte`, which provides a 4% performance gain.
- `-opt-streaming-cache-evict=0` is specific to MIC and used to generate streaming store instructions. This flag provides 8% gain for the direct force code on MIC.
- `-openmp` switch is necessary to run OpenMP code.

Details of the optimization flags used for tuning can be found in the compiler manual [35].

All benchmark systems were simulated in the isothermal-isobaric (NPT) ensemble with a 2fs time step and hydrogen bonds constrained using the SHAKE algorithm[16]. All benchmarks were run at a constant target temperature of 300K using a Berendsen weak-coupling thermostat[38] with a relaxation time of 1.0 ps. Factor IX and cellulose were kept at constant pressure using the Monte Carol barostat[32], which rigorously samples from the isobaric–isothermal ensemble and does not require the computation of the virial, whereas in the stmv system the Berendsen barostat[38] was used, which continuously scales the volume of the system by computing the virial, to move the pressure back toward the target pressure. All benchmark systems employed a direct space cutoff (factor IX(8Å), cellulose(8Å) and stmv(9Å)) and periodic boundary conditions with long-range electrostatics computed using the Particle Mesh Ewald method. Factor IX and cellulose were simulated for 2000 time steps and stmv was simulated for 1000 time steps. The input files used in the simulations are provided in the supporting documentation.

Factor IX did not see any performance improvement when executed on an Intel Xeon Phi coprocessor coupled to any of the CPUs specified above due to data transfer overheads. This serves as an indication of the minimum system size suitable for acceleration with an Intel Xeon Phi coprocessor using the *pmemd* offload approach. On 2 x Haswell (28 cores), the CPU-only version using 28 MPI tasks performs at $10.06 \text{ ns day}^{-1}$, whereas the MIC-offload version performs at 9.8 ns day^{-1} for FactorIX. Thus, the 90K atoms benchmark is a critical data point; workloads smaller than this will have slower performance when simulated with the MIC-offload implementation, whereas bigger systems than this should see performance improvements when running MIC-offload. This is due to the overheads involved in coupling a host CPU to a MIC card i.e. data transfer costs.

Figures 10 and 11 compare the performance in terms of ns day^{-1} of the cellulose and stmv benchmarks, respectively, using the varying hardware setups detailed above and in the figure captions. As can be seen from the figures, using optimized compiler flags with 2 Intel Xeon E5 2697 v2 processors provides a reasonable performance improvement (~50-70%) compared with using the baseline compiler flags. By offloading work to an Intel Xeon Phi 7120P coprocessor performance is improved by a further 30-35%, attaining 2.00 ns day^{-1} for a system of cellulose and 0.55 ns day^{-1} for a system of stmv compared with 1.53 ns day^{-1} and 0.41 ns day^{-1} on 24 IVB CPU cores, respectively. The performance improvement achieved from offloading to a MIC coupled to the latest HSW architecture is ~18% for the cellulose system and 25% for the stmv system. A larger performance improvement is seen for the system of stmv on both IVB and HSW setups due to the larger system size, which allows for more computation per MIC core and so more opportunity for hiding data transfer latencies and MIC memory access latencies. However, as the Xeon architecture becomes more powerful (28 cores for HSW as opposed to 24 cores for IVB) a smaller performance improvement is achieved i.e. 35% performance improvement for stmv when using a MIC card with the IVB architecture compared with a 25% performance improvement when using a MIC card with the HSW architecture.

Figure 10: Performance of Cellulose (408,609 atoms) measured in ns day⁻¹ using 2 Intel Xeon E5 2697 v2 (IVB) processors (2 x 12 cores), 2 Intel Xeon E5 2697 v3 (HSW) processors (2 x 14 cores), 2 x IVB+ Intel Xeon Phi 7120P (Phi) coprocessor (61 cores), 2 x HSW + Phi and NVIDIA K40 . “Baseline’ refers to compilation without tuning of the compiler optimization flags detailed in the text and ‘optimized’ refers to compilation with the tuned compiler optimization flags.

Figure 11: Performance of STMV (1m atoms) measured in ns day⁻¹ using 2 Intel Xeon E5 2697 v2 (IVB) processors (2 x 12 cores), 2 Intel Xeon E5 2697 v3 (HSW) processors (2 x 14 cores), 2 x IVB+ Intel Xeon Phi 7120P (Phi) coprocessor (61 cores), 2 x HSW + Phi and NVIDIA K40 . “Baseline’ refers to compilation without tuning of the compiler optimization flags detailed in the text and ‘optimized’ refers to compilation with the tuned compiler optimization flags.

5 Conclusions

In this research version 14 of the *PMEMD* molecular dynamics (MD) engine in the AMBER MD software package was extended to support offloading to an Intel Xeon Phi coprocessor. Work involved the implementation of a two-fold decomposition strategy that employs spatial decomposition of the system using MPI followed by thread-based decomposition within offloading MPI tasks using OpenMP. The approach used was to have between 1 and 4 MPI tasks offload 78% of their PME direct sum and neighbor list build workload to the MIC. Each offloading MPI task spawns OpenMP threads in a MIC partition to execute equally weighted portions of the workload. The results from the PME direct sum are then passed back to the host and accumulated into the full force array.

Performance tests revealed that once the algorithm had been tuned with compiler optimization flags it performs 1.43x and 1.62x faster on 2 Intel Xeon E5 2697 v2 processors (2 x 12 cores) + Intel Xeon Phi coprocessor (61 cores) when simulating systems of cellulose (408K atoms) and a system of satellite tobacco mosaic virus (stmv, 1,067K atoms), respectively, compared with running on 2 Intel Xeon E5-2697 v2 processors (2 x 12 cores) without the use of an Intel Xeon Phi coprocessor or compiler optimization tuning. The theoretical double precision peak performance of an Intel Xeon E5 2697v2 processor is 518.4 GFLOP/s. When coupled to an Intel Xeon Phi 7120P coprocessor a total of 1.7184 TFLOP/s of double precision performance is theoretically

1
2
3
4 achievable, translating to a theoretical speed up of 3.3x. However, *pmemd MIC offload*
5
6 sees an actual speed-up of 1.62x for stmv, which is significantly lower. The reasons for
7
8 this are thought to be due to a lack of code vectorization by the compiler because of
9
10 irregular loop structures and inefficient memory usage. Resolving these issues would
11
12 require significant code restructuring.
13

14
15 Future modifications to *PMEMD MIC offload* will be aimed at finding solutions to these
16
17 bottlenecks. One major change to the algorithm will be the implementation of a mixed
18
19 precision model. We predict that this will to provide significant performance
20
21 improvement by allowing double the number of simultaneous calculations by the SIMD
22
23 vector units. Other future revisions will include improving the dynamic load-balancing
24
25 scheme and adding to the amount of computation carried out on the Intel Xeon Phi.
26

27
28 The resulting software modifications have been released as updates to the AMBER 14
29
30 software suite and are automatically downloaded and applied during installation. The
31
32 code is available as patch files on the AMBER website

33
34 (<http://ambermd.org/bugfixes/14.0/update.5>;
35
36 <http://ambermd.org/bugfixes/14.0/update.9>).
37

38 **6 Acknowledgements**

39
40 This work was supported by NSF SI2-SSE grant (NSF-1148276) to RCW. RCW also
41
42 acknowledges funding through an unrestricted gift from Intel Corp.
43
44

45 **7 References**

- 46
47
48
49 1. S. Andre, Z. Pei, H.C. Siebert, O. Ramstrom, H.J. Gabius, *Bioorg. Med. Chem.*, 14
50 (2006) 6314-26.
51
52 2. A. Grossfield, *Biochim. Biophys. Acta*.1808 (2011) 1868-78.
53
54 3. M.J. Harvey, G. De Fabritiis, *Drug Discov. Today* 17(2012) 1059-62.
55
56 4. M.C. Zwier, L.T. Chong, *Curr. Opin. Pharmacol* 10 (2010) 745-52.
57
58 5. J.D. Durrant, J.A. McCammon, *BMC Biol.* 9(2011) 71.
59
60 6. A. Pere, F.J. Luque, M. Orozco, *Acc. Chem. Res.* 45(2012) 196-205.
61
62 7. J. Wereszczynski, J.A. McCammon, *Q. Rev. Biophys.* 45(2012) 1-25.
63
64 8. J. Sponer, X. Cang, T.E. Cheatham, *Methods* 57(2012) 25-39.
65

9. L.C. Pierce, R.Salomon-Ferrer, F. Augusto, C. de Oliveira, J.A. McCammon, R.C. Walker, J. Chem. Theory Comput. 8(2012) 2997-3002.
10. A.A. Skjevik, B.D. Madej, R.C. Walker, K.J Teigen, Phys. Chem. B, 116(2012) 11124-36.
11. Å.A. Skjevik, B.D. Madej, C.J. Dickson, Callum, K. Teigen, R.C Walker, I.R. Gould Chem. Comms., (2015).
12. M. Karplus, J.A. McCammon, Nat. Struct. Biol., 9(2002) 646-52.
13. S.A Adcock, J.A. McCammon, Chem. revs.,106 (2006) 1589-1615.
14. Y. Duan, P.A. Kollman, Science,282 (1998) 740-4.
15. J.L. Klepeis, K. Lindorff-Larsen, R.O. Dror, D.E Shaw, Curr. Opin. Struct. Biol., 19 (2009) 120-7.
16. J. P. Ryckaert, G.C. Herman, J. C. Berendsen, J, Comp. Phys., 23 (1977) 327-341.
17. D. Hamelberg, J. Mongan, J.A. McCammon, J. Chem. Phys., 120(2004) 11919-11929.
18. Y. Sugita, Y. Okamoto, Chem. Phys. Lett., 314 1(999) 141-151.
19. X. Wu, B.R. Brooks, Chem. Phys. Lett., 381 (2003) 512-518.
20. J. Schlitter, M. Engels, P. Kruger, J. Mol. Graphics, 12 (1994) 84-89.
21. J. Schlitter, M. Engels, P.Krüger, E.Jacoby, A. Wollmer, 10 (1993) 291-308.
22. K. J. Bowers, E.Chow, X. Huageng, R.O. Dror, M.P. Eastwood, B.A. Gregersen, J.L. Klepeis, I. Kolossvary, M.A Moraes, F.D. Sacerdoti, J.K. Salmon, S. Yibing, D.E. Shaw, in SC 2006 Conference, Proceedings of the ACM/IEEE. 2006.
23. R. Salomon-Ferrer, A.W.Götz, D. Poole, S. Le Grand, R.C. Walker, J Chem. Theory Comput., 9 (2013) 3878-3888.
24. S. Le Grand, A.W. Götz, R.C. Walker, Comput.. Phys. Commun., 184(2013) 374-380.
25. A.W. Götz, M.J. Williamson, X. Dong, D. Poole, S. Le Grand, R.C. Walker, J Chem. Theory Comput., 8 (2012) 1542-1555.
26. D.A. Case, V.B., J.T. Berryman, R.M. Betz, Q. Cai, D.S. Cerutti, T.E. Cheatham, III, T.A. Darden, R.E. Duke, H. Gohlke, A.W. Goetz, S. Gusarov, N. Homeyer, P. Janowski, J. Kaus, I. Kolossváry, A. Kovalenko, T.S. Lee, S. LeGrand, T. Luchko, R. Luo, B. Madej, K.M. Merz, F. Paesani, D.R. Roe, A. Roitberg, C. Sagui, R. Salomon-Ferrer, G. Seabra, C.L. Simmerling, W. Smith, J. Swails, R.C. Walker, J. Wang, R.M. Wolf, X. Wu and P.A. Kollman, *AMBER 14*. 2014: University of California, San Francisco.
27. Walker, R. *AMBER 14 NVIDIA GPUACCELERATION SUPPORT*. Benchmarks 2014 [accessed 2014 February]; Available from: <http://ambermd.org/gpus/benchmarks.htm> - Benchmarks.
28. R. Meyer, Phys. Rev. E, 88(2013) 053309.
29. S.J. Pennycook, C.J. Hughes, M. Smelyanskiy, S.A. Jarvis, Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. 2013. IEEE.
30. M. Plotnikov, *GROMACS for Intel® Xeon Phi™ Coprocessor*. 2014 [accessed 2015 February]; Available from: <https://software.intel.com/en-us/articles/gromacs-for-intel-xeon-phi-coprocessor>.

31. X. Sun, *Molecular Dynamics Optimization on Intel® Many Integrated Core Architecture (Intel® MIC)*. 2013 [accessed 2015 February]; Available from: <https://software.intel.com/en-us/articles/molecular-dynamics-optimization-on-intel-many-integrated-core-architecture-intel-mic?language=en>.
32. M.P. Allen, D.J. Tildesley, *Computer simulation of liquids*. 1989, Clarendon Press. 110-135.
33. W.D. Cornell, P. Cieplak, C.I. Bayly, I.R. Gould, K.M. Merz, D.M. Ferguson, D.C. Spellmeyer, T. Fox, J.W. Caldwell, P.A. Kollman, *J. Am. Chem. Soc.*, 117 (1995) 5179-5197.
34. T. Darden, D. York, L. Pedersen, *J. Chem. Phys.*, 98(1993) 10089-10092.
35. *User and Reference Guide for the Intel® Fortran Compiler 14.0*. 2014 [accessed 2015 February]; Available from: https://software.intel.com/en-us/compiler_14.0_ug_f.
36. Intel. *Intel® MPI Library User's Guide for Linux* OS*. 2014 [accessed 2015 February]; Available from: Intel® MPI Library User's Guide for Linux* OS.
37. Intel. *Intel® Manycore Platform Software Stack (MPSS)*. 2014 [accessed 2015 February]; Available from: <https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss>.
38. H.J. Berendsen, J.P.M Postma, W.F. van Gunsteren, A.R.H.J. DiNola, J.R. Haak, *J.Chem. Phys.*, 81(1984) 3684-3690.

Figure 1

Figure 2

Figure 3

Figure 4

Figure 5

Figure 6

Figure 7

Figure 8

Figure 9

Figure 10

Figure 11