Performance enhancement of Ionic structure in liquids confined by dielectric interfaces (A nanoconfinement framework)

Kadupitiya Kadupitige, UID: 2000253911, Uname: jasakadu
Final Report for Research Assistantship in Fall semester 2017

Abstract— **Computationally intensive simulations that use complex molecular dynamics (MD) concepts to propagate the ions are required to determine the ionic structure of nanoparticles confined between surfaces. Nanoconfinement simulation framework designed by Jadhao et al. [1] is one such simulation framework allowing worldwide researchers to observe the density profiles and behavior of nanoparticles. This research presents a hybrid implementation that utilizes MPI for distributed memory parallelism and OpenMP for shared memory parallelism for performance enhancement of the nanoconfinement framework. Several memory optimization techniques were adopted to reduce the computational time of the framework. Implementation details of MPI, OpenMP, and Hybrid MPI/OpenMP parallelization of the nanoconfinement simulation framework are provided. Using the hybrid approach and memory optimization techniques, computation time was reduced from 12 hours to 13.25 mins yielding a maximum speedup of 54.34.**

I. Problem statement

The ionic structure of nanoparticles confined between biological and synthetic surfaces determines the outcome of many synthetic and biological systems such as colloidal dispersions, emulsions, hydrogels, DNA, cell membranes and proteins [1]. These nanoparticles can be considered as salt ions in most of the biological systems [1]. Understanding the behavior of ionic structures in such systems are crucial for the enhancement of many industrial applications such as double-layer supercapacitors for energy storage and advance formulation science for protein-based pharmaceutical preparations and cellular therapies [1, 2]. Figure 1 illustrates an ionic structure in liquids confined by dielectric interfaces.

Jadhao et al. [1] has created a simulation framework (Nano confinement framework) and a science gateway (Nano confinement gateway) to experiment and investigate a different kinds of ionic structures and environmental attributes including ion valency and salt concentration using standard and advanced molecular dynamics (MD) methods for non-polarizable surfaces and dielectric interfaces respectively. This framework is written in C++ programming language by adopting the sequential programming model. These standard MD simulations are computationally intensive, and it takes nearly 12 hours to run a simulation for one million steps (1 Nano second of real time MD) with 422 nanoparticles using standard set of parameters to get the density profile of the ionic structure [1, 3]. Currently, the framework (the science gateway) is being used by different individuals all over the word for educational purposes for fewer ions and less number of steps. Hence, it is necessary that the simulation time be reduced in order to enable particle dynamics researchers to carry out advance MD simulations as well as to allow more particle dynamics researchers to engage with the Nano confinement framework.

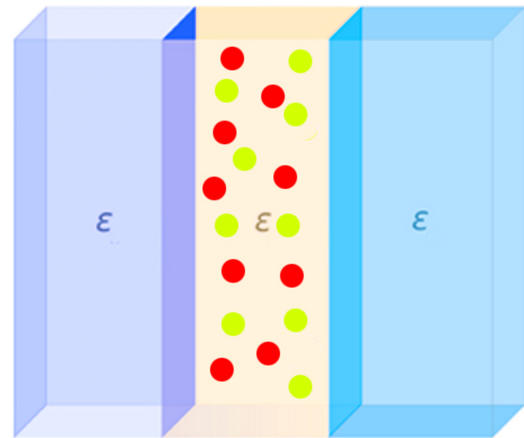

Fig. 1. An Example for Ions (positive-blue and negative-green) distribution in liquid confined by two dielectric interfaces

In this research, the possibilities of performance optimization of existing Nano confinement framework is studied using shared memory parallelization, distributed memory parallelization and the hybrid approach using both shared memory and distributed memory.

II. Literature review

The techniques used for parallelizing different kind of simulation frameworks or applications could be broadly categorized into three approaches; shared memory model (OpenMP), distributed memory model (MPI) and hybrid model (OpenMP and MPI) [4, 5]. OpenMP, MPI, and hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes, have been in use for many years [6, 7, 8, 9, 10]. This study uses these techniques, and hence, they are discussed in this section.

Rabenseifner et al. [6] have investigated a comprehensive evaluation of different performance improvement factors and degradation factors in high-performance computing (HPC) with respect to a modern hierarchical hardware design. Due to the reduced communication and memory consumption, or improved load balance evident in hybrid (MPI plus OpenMP), this research has employed pure Message Passing Interface (MPI), pure OpenMP and hybrid (MPI plus

OpenMP) to pinpoint the cases where a hybrid programming model could be the most effective solution. They claim that parallel programming model should consider combining distributed memory parallelization (on the node interconnect) with shared memory parallelization (inside each node) [6].

Recent research done by Mahinthakumar et al. [7, 8] have applied parallel programming approach for an implicit finite-element methodology in groundwater transport simulations using hybrid MPI-OpenMP programming model. They have used a domain decomposition strategy to decompose their original program to enable parallelization for distributed memory model using MPI [7, 8]. Researchers have implemented straightforward loop-level parallelism with several loop modifications using OpenMP directives inside each MPI process to enable shared memory model as they tested their simulations in symmetric multiprocessing (SMP) nodes [7]. Parallel performance results were compared using four different architectures and researchers claim that the hybrid approach outperformed both pure MPI and pure OpenMP performance by giving promising results when using it with SMP cluster architectures [7].

Tang et al. [9] have also used OpenMP, MPI hybrid approach for groundwater model calibration using multi-core computers. According to the researchers, the computational model for groundwater calibration utilizes between one hundred to one thousand forward solutions, each entailing many nonlinear partial differential equations, thereby leaving a computationally intensive problem to solve [9]. First, they profiled the sequential program using GPROF and identified a single parallelizable loop account for over 97% of the total computational time. Researchers have adopted the OpenMP programming model for the identified parallelizable loop and the MPI programming model for parallelizing the Jacobian calculation and lambda search in their ground water calibration algorithm [9]. Reported results indicate that the calibration time was reduced from weeks to a few hours by using this hybrid approach in 100200 compute cores [9].

A research done by Mininni et al. [10] focused on performance improvements of Pseudospectral computations for fluid turbulence using MPI for distributed memory parallelism and OpenMP for shared memory parallelism. Their approach focused on achieving exceptionally high Reynolds numbers in pseudospectral computations of fluid turbulence in massively parallel processing systems [10]. Researchers have used domain decomposition techniques to achieve numerical discretizations of the problem to implement the hybrid parallel programming approach on top of the original sequential program. They claim that the hybrid methodology provides good scalability up to 20,000 compute nodes with a maximum efficiency of 89%, and a mean of 79% [10]. Furthermore, they have shown that the cost of communication increased with the number of MPI processes, and the hybrid scheme allowed to reduce the number of MPI processes by utilizing OpenMP programming model. Researchers further claim that for a given workload per MPI task, the hybrid scheme would clearly outperform the pure MPI version in terms of speed up and computation time [10].

As described above, most of the parallel programming techniques applied to simulations and science applications are focused on pure MPI, pure OpenMP and hybrid OpenMp-MPI programming models. Moreover, hybrid techniques have given comparable, or even better results than pure shared memory(OpenMP) or pure distributed memory (MPI) approaches in most cases [7, 8, 10]. There are also many researches which are focused on performance enhancement of different simulations. Based on the review of the literature, there has not been a single study that has focused on performance enhancement of simulation frameworks for ionic structure in liquids confined by dielectric interfaces. Due to these facts, this research will also employ the three approaches mentioned above in order to achieve performance enhancement of the Nano confinement framework.

III. Applications of Nanoconfinement framework

Nanoscale systems and materials are hard to model using real world experiments. Modeling and simulation are the best chance to understand and draw inferences about the real system [1]. The main applicational use of this framework is designing and experimenting on new soft materials by simulating soft matter systems in drug delivery, soft electronics, and energy. Example applications are super capacitors and synthetic red blood cells [1].

IV. Methodology

A. Overview of Nano confinement serial algorithm

The nanoconfinement code used in this research was developed by Jadhao et al. [1] to simulate ions confined by the dielectric interfaces using MD methods. This program assumes infinite extension of the interfaces by using periodic boundary conditions in both x and y directions in the 3D space [1]. The program also considers the system of ions and liquid confined within the two surfaces as a rectangle box with volume V while maintaining periodic boundaries as shown in figure 1 and equation 1 [1].

$$V = H * L * L \tag{1}$$

where H=3 nm, L= length of the interface along the x and y directions.

The total number of ions that associate with the simulation is defined as N, which is a combination of negative ions and positive ions $(N_N + N_P)$. The ions are modeled as soft sphere of radius σ ($\sigma = l_B/2 = 0.357nm$, where l_B is the Bjerrum length in water at room temperature) interact via a purely repulsive Lennard-Jones (LJ) potential. The interactions between ions and interfaces are also considered as purely repulsive LJ potential. The ions also interact with each other using the Coulomb potential energy as given in the equation 2 [1].

$$U_c = \frac{q_i * q_j * l_B}{r_{ij}} \tag{2}$$

Where q_i and q_j are the charges on the ion i and ion j respectively, and r_{ij} is the distance between considered ions.

Researchers have assumed that the liquid between the confined plates as water and all simulations are performed at room temperature (298 K) in all their experimental simulations. Since the experiments are in nanoscale, reduced time unit (τ) has been used throughout the simulations as shown in equation 3 [1].

$$\tau = \{\frac{m\sigma^2}{k_B T}\}^{\frac{1}{2}} \tag{3}$$

where m is the ionic mass, σ is the radius of ion, k_B is the Boltzmann constant and T is the room temperature.

Velocity verlet algorithm [11] has been used for ionic propagation through the medium with a timestep of $\Delta t = 0.0005\tau$. Researchers who have worked in nanoconfinement framework have found that 1 million-time steps are the least number of steps in which the system reaches an equilibrium [1].

The pseudocode of the nanoconfinement program is shown in figure 2 [1]. The program starts with initializing and calculating various important parameters relevant to MD. Ionic propagation starts with pre-calculated parameters. For each step, it first updates the velocity for half time, then it updates the positions of the ions. Next it calculates the forces on each ion and finally it updates the velocity for the next half time. It provides the ionic structure of the system when all the time steps are computed.

```
[f, v, r, ~] = mdInitialization ();
for time steps from 1 to 1,000,000
        v = updateVelocity(f, v , Δt/2 , ~)
                v (t + Δt/2) ← v(t);
        r= updatePositions(r, v, Δt, ~)
                r (t + Δt) ← r(t);
        f= mdCalculateForce (~);
        v= updateVelocity(f, v, Δt/2, ~)
                v (t + Δt/2) ← v(t);
computeEnergy ();
computeDensityProfile ();
```

Fig. 2. Pseudocode of the nanoconfinement program

B. Inputs/Outputs of the program

Inputs to the program can be divided in two categories; physical parameters and computational parameters. Figure 3 shows the inputs of the program. The program produces the ionic structure in the form of density profile at the end of the simulation. The density profile for different input parameters is shown in figure 4. It also provides energy distribution, temperature distributions, ion positions with respect timestamp, etc.

Physical parameters

-X, -Y, -Z : Box dimension in nanometers, default values are (11.424, 11.424, 3).

-p, -n : Positive and negative valency inside, default values are (1,-1).

-c : Salt concentration inside, default value is 0.1.

-d : Salt ion diameter inside, default value is 0.714.

-g : Interface discretization width, default value is 4.

-e, -E dielectric constants inside and outside, default values are (80.1, 80.1).

-T time step value in md 0.0005

Computational parameters

-S : Time steps used in simulation, default value is 1,000,000.

-P : Density profile production begin value, default value is 100,000.

-F : Sampling frequency for density profile, default value is 100.

-x: Compute additional values for position, energy, etc., default value is 1000.

-w : Write density interval value, default value is 100,000.

-m : Snapshot saving parameter for movie, default value is 1000.

Fig. 3. Inputs for the nanoconfinement program

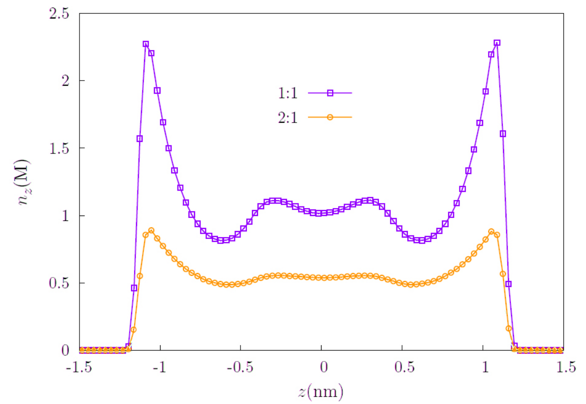

Fig. 4. Density profile of positive ions for an electrolyte (at C = 0.1) confined within two non-polarizable planar interfaces.

C. Performance considerations/Issues

As explained in the problem statement, the standard MD nanoconfinement simulation takes approximately 12 hours to run a simulation for one million steps using the standard set of parameters to obtain the density profile of the ionic structure. Even though the nanoconfinement framework has been deployed as a science gateway, it is not used for actual research experiments due to the fact that the runtime is high even for a simulation of smaller scales (i.e. with the standard set of parameters).

The sequential program was evaluated for performance profiling in order to determine where it is spending its time. The nanoconfinement program was tested in the Indiana university BIGRed2 cluster, which features a hybrid architecture based on two Cray Inc., and it runs a proprietary variant of Linux called Cray Linux Environment [12]. The performance profiling was done using the Performance Counters for Linux

(PERF) [13] and figure 5 shows the output obtained from the perf report. This report clearly shows that the nanoconfinement program was spending 64.33% of its total computation time on calculating the forces between the ions for each time step of the simulation (see mdCalculateForce function in figure 2 and formdcalculateforce in figure 5). The report also revealed that no other major function call takes considerable computation time compared to force calculation. The second noticeable computation time was taken by object creation procedures, whereas the third highest computation time was consumed by basic vector operations such as addition, subtraction and scalar multiplication.

There are three major tasks in the nanoconfinement program such as update position, update velocity and force calculation. The time complexities for update position, update velocity and force calculation are $O(n)$, $O(n)$, $O(n^2)$ respectively. Performance profiling results are verified if the time complexities of those sub routines are considered. As a result, this study focused on optimization and parallelization of the force calculation subroutine.

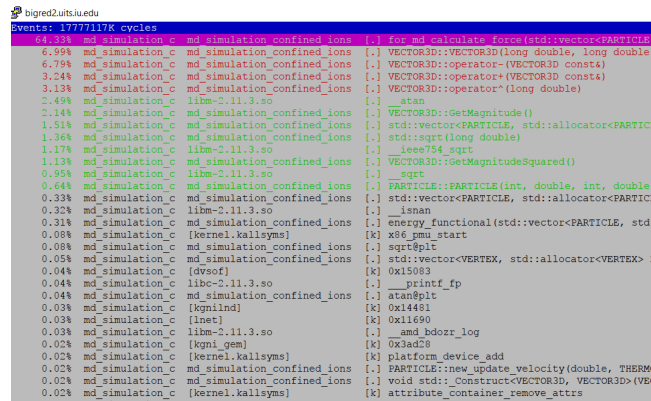

Fig. 5. Distribution of computational time among subroutines for the nanoconfinement program

D. Approach taken to parallel algorithm version

According to the performance profiling, the force calculation subroutine was identified as the most time-consuming function. It also found to be the only function which had a time complexity of $O(n^2)$ for a single time step iteration. The force on an ion is a combination of five force elements: force due to coulomb potential energy - $O(n^2)$, force calculated for purely repulsive LJ potential - $O(n^2)$, force due to interactions with left wall - $O(n)$, force due to interactions with right wall - $O(n)$ and safe force calculation for dummy particles considering both walls - $O(n)$.

This study employs three parallelization approaches. The distributed memory approach was tested with MPI. The shared memory parallelization was tested with OpenMp. For the third approach, the hybrid model, parallelization was achieved by combining MPI and OpenMP approaches.

1) Distributed memory parallelization with MPI: The MPI implementation of nanoconfinement program uses one-dimensional (1D) problem decomposition considering ions position vector. The nanoconfinement program propagates by calculating subroutines such as first half velocity, position, force and the second half velocity for each iteration. It is not effective to decompose the problem for all these subroutines as the highest computation time is consumed by the force calculation. The MPI communication calls and the overhead per iteration will increase if the problem is decomposed for all subroutines. Figure 6 shows the procedure applied to enable the distributed memory parallelization with MPI. Figure 7 is the pseudocode for calculating MPI discretization parameters (boundary conditions) for the problem decomposition. In this approach, the simulation propagation happens sequentially in every process until each program inside the processes needs to calculate the current forces based on the ion location and other MD parameters. Inside each of these processes, the force (partial force calculation) calculation happens only for the boundary parameters defined for that particular process. After the partial force calculation has been completed as explained in the figure 6, the MPI collective operation MPIAllgather was used to distribute the partial data of the forces among all processes. Subsequently, each process calculates other required MD parameters inside the process and move on to perform the next iteration in the propagation simulation.

2) Shared memory parallelization with OpenMP: Even though the shared memory parallelization approach with OpenMP is based on a state of art loop level parallelism, few modifications were made to the computationally intensive loops inside the force calculation procedure so as to improve the shared memory parallelization. As explained in the sequential algorithm section, the force calculation contains five other sub routines which contained nested for loops causing the time complexity of $O(n^2)$. Firstly, the straightforward loop level parallelism is achieved by implementing OpenMP compiler directives on all nested loops. The OpenMP parallelization is applied at the outermost loops to reduce OpenMP overheads, such as thread generation and data copy. The dynamical distribution and the ordering of the loop index from a large to a small task are applied to obtain better load balancing. Several memory optimization techniques such as using C-language arrays instead of C++ vectors (as it favors data locality) were used to improve the efficiency of the program [14]. Repetitive memory allocations inside the force calculation procedure were moved to the outside of the force calculations as force calculation routines had time complexity of $O(n^2)$. Figure 8 shows the OpenMp parallelization applied to the force calculation subroutine due to coulomb potential energy.

As these $O(n^2)$ force calculation subroutines contained repetitive force calculation between ions, a force matrix calculation approach was also tried in order to reduce the time complexity to $O(n^2/2)$. Instead of accumulating all the force elements for one outer loop iteration, the force matrix was calculated only for the upper diagonal element of the matrix. This approach was successful because the force calculation equation was computationally time consuming as it contained square root of vector elements.

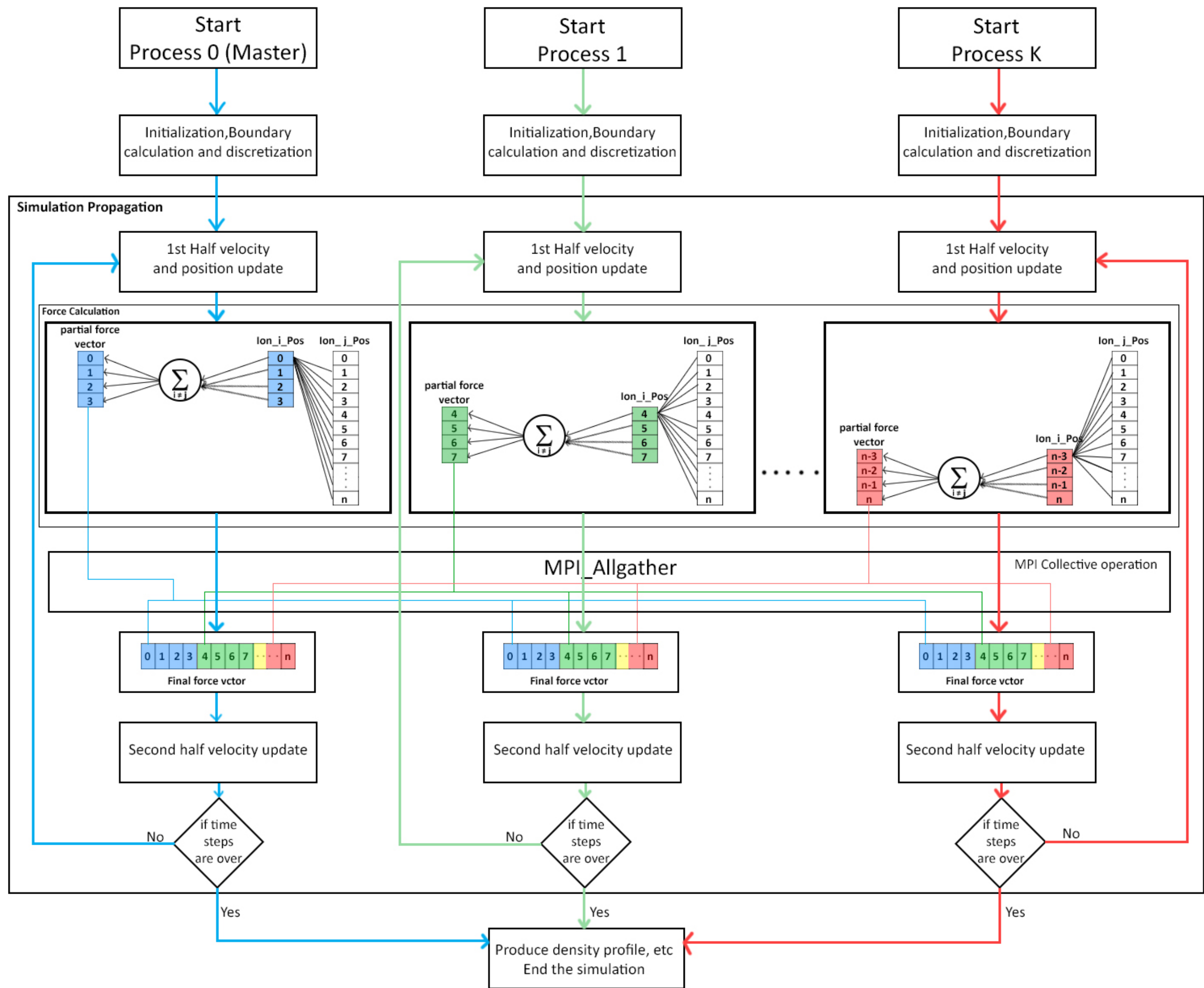

Fig. 6. Distributed memory parallelization approach with MPI

```
range = problemSize / numberOfProcesses + 1.5;

lowerBound = processID * range;

upperBound = (processID + 1)*range - 1;

   if (processID == numberOfProcesses - 1)

      upperBound = problemSize - 1;

   if (numberOfProcesses == 1)

      lowerBound = 0;

      upperBound = problemSize - 1;
```

Fig. 7. Pseudocode for calculating MPI discretization parameters (boundary conditions)

```
#pragma omp parallel for schedule(dynamic) default(shared) private(i, j, .....)
for (i = 0; i < ion.size(); i++) {
        parameterInitialization();
        for (j = 0; j < ion.size(); j++) {
          CPEForceCalculation();
        }
        forceVectorScaling();
}
```

Fig. 8. OpenMp parallelization applied for force calculation subroutine due to coulomb potential energy

3) Hybrid MPI/OpenMP parallelization: Following Rabenseifner et al. [6], the hybrid masteronly model was tested by combining the distributed memory MPI approach and the shared memory OpenMP approach. The hybrid masteronly model uses one MPI process per node and OpenMP on the cores of the node, with no MPI calls inside parallel regions. This hybrid model enables the domain decomposition under a two-level mechanism. This approach is applied for the force calculation subroutine in the nanoconfinement program. On the MPI level, a coarse-grained domain decomposition is performed using boundary conditions as explained in figure 6 and 7. The second level of domain decomposition is achieved through OpenMP loop level parallelization inside each MPI process. This multilevel domain decomposition has advantages over pure MPI or pure OpenMP, when cache performance is taken into consideration. This strategy also provides the maximum access locality, a minimum of cache misses, non-uniform memory access (NUMA) traffic and inter-node communication [6].

V. Results and Discussion

This section explains the application performance of OpenMP, MPI, and hybrid parallel programming models on nanoconfinement program. Nanoconfinement program was benchmarked using BigRed II cluster nodes (Maximal achieved performance of 596.4 teraFLOPS, features a hybrid architecture based on two Cray, Inc., 344 XE6 (CPU-only) compute nodes and 676 XK7 "GPU-accelerated" compute nodes, providing a total of 1,020 compute nodes, 21,824 processor cores, and 43,648 GB of RAM. Each XE6 node has two AMD Opteron 16-core Abu Dhabi x8664 CPUs and 64 GB of RAM; each XK7 node has one AMD Opteron 16-core Interlagos x8664 CPU, 32 GB of RAM, and one NVIDIA Tesla K20 GPU accelerator) [15].

As explained in the problem statement section, the standard MD simulation takes approximately 12 hours to complete a simulation using the default parameter set defined in figure 2. Using the memory optimization techniques discussed in OpenMP shared memory parallelization, the run time of the simulation was reduced to approximately 7 hours. The following sub sections describe the performance results for each approach tested in this research. All the experiments are done with the parameter set defined in figure 2.

A. Pure MPI Performance

Figure 9 shows strong scaling plot of pure MPI performance with 424 ions and 1 million time steps in nanoconfinement program. Even though the Ideal strong scaling graph should increase the speedup as the number of processes increases, it seems that the nanoconfinement program strong scale well up to 48 processes with maximum speedup of 16.80. This may be due to the overhead of running multiple processes on 424 ions (1 Million time steps) is higher than the parallelization achieved through the MPI model.

Table 1 shows the weak scaling of pure MPI performance with different number of ion sizes and 1 million time steps. From table 1, it is clear that the problem size is proportional to the MPI processes which are used to achieve the maximum speedup. The maximum speedup of 41.86 was obtained for 512 MPI processes when program is executed with 4242 ions.

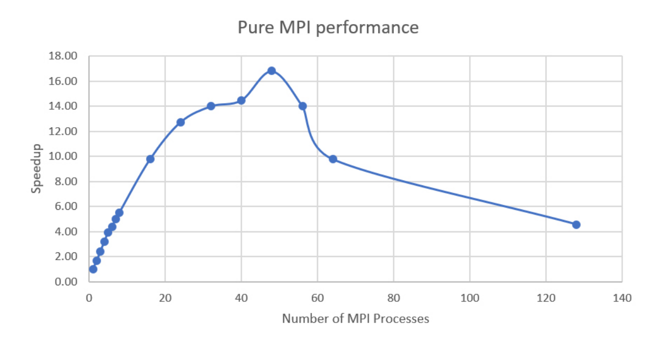

Fig. 9. Strong scaling plot of pure MPI performance with 424 ions and 1 million time steps.

TABLE I

Computational Time (minutes) and Speedup (in parentheses) of pure MPI performance with 1 million time steps.

# Procs	64	128	256	512
Ions=424	**51(13.21)**	55(12.25)	60(11.23)	81(8.32)
Ions=848	134(26.20)	**123(28.55)**	130(27.02)	178(19.73)
Ions=1272	250(24.00)	**222(27.03)**	228(26.32)	274(21.90)
Ions=1696	431(23.22)	363(27.58)	**351(28.52)**	421(23.78)
Ions=2120	634(23.51)	524(28.44)	**492(30.29)**	572(26.06)
Ions=4242	2378(26.5)	1843(34.2)	1643(38.4)	**1509(41.8)**

B. Performance of Pure OpenMP

Figure 10 shows the strong scaling plot of the performance of pure OpenMp O(n2) with 424 ions and 1 million time steps in nanoconfinement program. It is clear that the speedup converges at around 10.7 with 16 OpenMP threads. The speedup increased only up to 16 threads. Since this benchmarking was done using BigRed II 676 XK7 "GPU-accelerated" compute nodes which have 16 internal cores per node, the strong scaling results were in line with the expectations. It seems that GPU node is being oversubscribed when more than 16 threads are used.

Figure 11 shows the strong scaling plot of the performance results of pure OpenMp $O(n^2/2)$ with 424 ions and 1

million time steps in the nanoconfinement program. Using this approach, the sequential program runtime reduced from 418 mins to 350 mins. The maximum speedup of 11.30 was obtained using this approach. However, the speedup improvement was not significant in comparison to the $O(n2)$ approach. This was due to the fact that the force matrix procedure used in the $O(n^2/2)$ approach contained many 2D array accesses and writes compared to the previous 1D array force vector in the $O(n^2)$ approach.

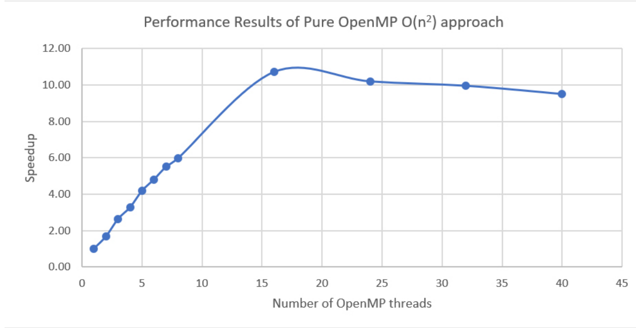

Fig. 10. Strong scaling plot of the performance results of the pure OpenMp $O(n^2)$ approach, with 424 ions and 1 million time steps.

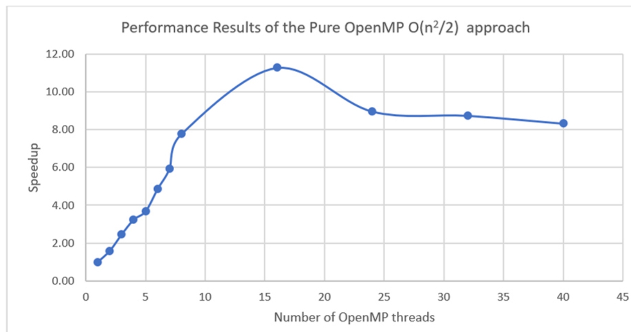

Fig. 11. Strong scaling plot of the performance results of the pure OpenMp $O(n^2/2)$ with 424 ions and 1 million time steps.

C. Performance of Hybrid MPI/OpenMP

Figure 12 shows the strong scaling plot of the performance of the hybrid approach with 424 ions and 1 million time steps in the nanoconfinement program. From the experimental results, Rabenseifner et al.s [6] claim that allocating all the internal cores for OpenMP thread was validated and the optimum number of OpenMP threads was selected to be 16 for any number of MPI processes since the experiment was carried out in BIGRed IIs GPU nodes. Using the hybrid methodology program, the runtime was reduced from 418 mins to 13.25 mins. The maximum speedup of 31.62 was obtained when the experiment was carried out with 256 processes (16 MPI nodes and 16 OpenMP threads inside each MPI node). This maximum speedup was calculated without considering the execution time reduction gained from the memory optimization techniques. The hybrid methodology was also able to reduce the execution time from 69.5 hours to 2.25 hours with a speedup of 30.88 when the nanoconfinement program was executed for 10 million time steps with 424 ions.

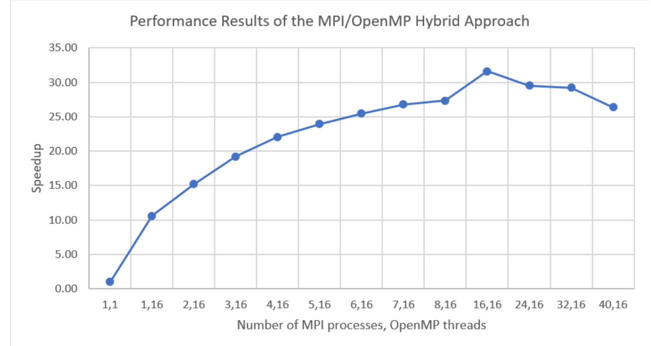

Fig. 12. Strong scaling plot of the performance of MPI/OpenMP hybrid with 424 ions and 1 million time steps.

VI. Conclusion

This research considered pure MP, pure OpenMP and hybrid MPI/OpenMP models for a nonoconfinement simulation framework. The pure MPI parallelization is based on domain decomposition and the pure OpenMP parallelization is based on loop level directives. The hybrid MPI/OpenMP model utilizes one MPI process per node and OpenMP on the cores of the node, with no MPI calls inside parallel regions. Several memory optimization techniques were used to reduce the execution time from 12 hours to 7 hours. The maximum speedup obtained using pure MPI approach was 16.80 with standard MD parameters. The pure MPI approach works better when there are more number of ions associated with the problem. The pure MPI approach was able to achieve a maximum speedup of 41.86 when tested with 4242 ions. Using the pure OpenMp $O(n^2)$ and $O(n^2/2)$ approaches, the maximum speedup of 10.7 and 11.30 were obtained under the standard default MD settings. The hybrid approach achieved a maximum speedup of 31.62 with the best execution time of 13.25 mins with 424 ions and 1 million time steps along with other standard MD parameters. The hybrid MPI/OpenMP model has given the best performance result out of the three approaches by reducing the execution time of the program from 418 mins to 13.25 mins.

VII. References

[1] Jing, Y., Jadhao, V., Zwanikken, J. W., and Olvera de la Cruz, M. (2015). Ionic structure in liquids confined by dielectric interfaces. The Journal of chemical physics, 143(19), 194508.

[2] Elliott, Gloria D., Regina Kemp, and Douglas R. MacFarlane. ”The Development of Ionic Liquids for Biomedical Applications Prospects and Challenges.” 2009. 95-105.

[3] Development of the Nanoconfinement Science Gateway

[4] Dagum, L., and Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. IEEE computational science and engineering, 5(1), 46-55.

[5] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. Parallel computing, 22(6), 789-828.

[6] Rabenseifner, R., Hager, G., and Jost, G. (2009, February). Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on (pp. 427-436). IEEE.

[7] Mahinthakumar, G., and Saied, F. (2002). A hybrid MPI-OpenMP implementation of an implicit finite-element code on parallel architectures. the International Journal of High Performance Computing Applications, 16(4), 371-393.

[8] Mahinthakumar, G., and Sayeed, M. (2005). Hybrid genetic algorithmlocal search methods for solving groundwater source identification inverse problems. Journal of water resources planning and management, 131(1), 45-57.

[9] Tang, G., DAzevedo, E. F., Zhang, F., Parker, J. C., Watson, D. B., and Jardine, P. M. (2010). Application of a hybrid mpi/openmp approach for parallel groundwater model calibration using multi-core computers. Computers and Geosciences, 36(11), 1451-1460.

[10] Mininni, P. D., Rosenberg, D., Reddy, R., and Pouquet, A. (2011). A hybrid MPIOpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence. Parallel Computing, 37(6), 316-326.

[11] Martys, N. S., and Mountain, R. D. (1999). Velocity Verlet algorithm for dissipative-particle-dynamics-based models of suspensions. Physical Review E, 59(3), 3733.

[12] Rodgers, G. P., and Stewart, C. A. (2006). On the road to petascale processing with IUs Big Red Supercomputer and IBM BladeCenter H.

[13] Eranian, S. (2006, July). Perfmon2: a flexible performance monitoring interface for Linux. In Proc. of the 2006 Ottawa Linux Symposium (pp. 269-288).

[14] Acklam, E., Jacobsen, A., Langtangen, H. P., and Bruaset, A. M. (1998). Optimizing C++ code for explicit finite difference schemes.

[15] Indiana University Indiana University Indiana University. (n.d.). Retrieved November 12, 2017, from https://kb.iu.edu/d/bcqt