Brownian Dynamics with Steric and Hydrodynamic Interactions CILK Plus vs OpenMP

Harsh Shrivastava GTID: 903241811

HPPC Final Project, Github link

Abstract

I implemented the complete Brownian Dynamics (BD) code including both, the steric and hydrodynamic interactions in shared memory. Cholesky decomposition was used in the process of updating the particle positions at every iteration. The objective of this project was to compare the 2 parallel programming paradigms, namely CILK Plus and OpenMP. Comparison of runtimes of partwise implementation, speed-up graphs, vectorization results, ease of programming are all included in this report. The complete BD was visualized in 3D using 'Visual Molecular Dynamics' tool.

1. Introduction and Overview

The skeleton code provided in project1 and project2 were used and the Brownian Dynamics (BD) code was built upon them. Multiple optimizations were done to speed up the code and will be discussed in the following sections. Roughly, the implementation can be divided into 3 parts:

1. Calculating the Steric Interactions between particles
2. Calculating the Hydrodynamic Interactions between particles
3. Using Cholesky Decomposition to combine both interactions and update the particle positions

In the following sections, I will go through the implementation details of each part and show comparison results between OpenMP and Cilk Plus, with and without vectorization. After that a section will be on details of overall Brownian Dynamics runtimes and speed-up graphs. Finally, some discussions for further improvements and conclusions will follow.

Final Project for CSE 6230

Table 1. The results are for N= 10,000 particles, a = 1 (radius of particle), $\phi = 20\%$ vol fraction. The initial positions are initialized using uniform random function. Input file used: my-input-npos-10000.xyz and each interval= 1000 timesteps.

Algorithm	Diffusion Coefficient	Num of intervals	Time per time steps (secs)	Num cores
bd-fast	0.750536	10	0.000774542	30
	0.711228	100	0.000731972	30
bd-cell	0.750231	10	0.00108854	30
	0.715235	100	0.00109404	30

2. Steric Interactions

The project1 framework was used for this part. I tried various techniques to parallelize the interaction function and also came up with a faster serial code for calculation for steric interactions. In this report, I'll show the results of 2 of my fastest parallel implementations. These are:

1. bd-fast.c
2. bd-cell.c

Note: The codes for these implementations are in git. All results shown are obtained by running simulations on ***mic2***

For the above two implementations, I'll first demonstrate their correctness by verifying the diffusion coefficient value obtained and then briefly explain the techniques used for optimizations.

2.1. Diffusion Coefficient verification

Table1 shows that both the 'bd-fast' and 'bd-cell' algorithms give expected values of diffusion coefficient, thereby verifying the correctness of implementation. The results shown are for OpenMP implementation, similar results were obtained for the Cilk counterpart too.

2.2. Comparison on MIC

The figure1 and figure2 shows the runtime comparisons of OpenMP and Cilk Plus on 'bd-fast' and 'bd-cell' algorithm, respectively. We can observe here that the runtimes of OpenMP are better than Cilk Plus. Though, it was easier

to program in Cilk Plus than dealing with many different 'pragma' settings of OpenMP and also maintaining a list of 'shared' and 'private' variables. Cilk gave better results for 'bd-cell' algorithm than OpenMP, whereas in general I found OpenMP to outperform Cilk.

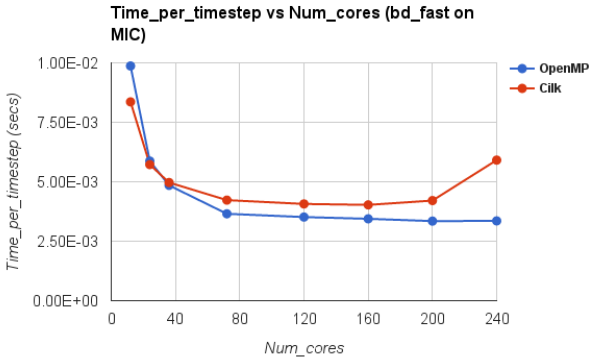

Figure 1. Comparison of time per timestep for N=10000 particles, between OpenMP and Cilk Plus versions. Results are on mic2 machine and running the bd-fast algorithm. Simulations were run using KMP_AFFINITY=verbose,granularity=fine,scatter/compact/balanced.

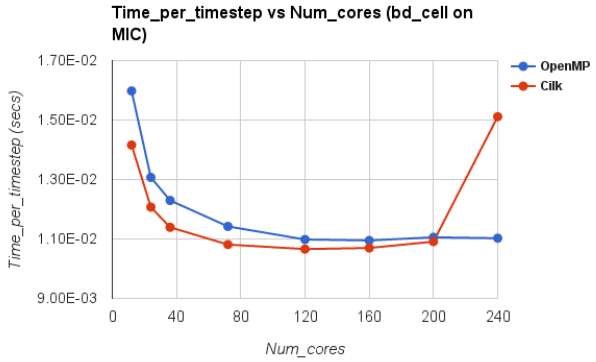

Figure 2. Comparison between OpenMP vs Cilk Plus versions. Results are of the bd-cell algorithm with remaining settings same as for figure1.

2.3. Optimized 'interactions.c' function

The following are the changes done to 'interactions.c' function provided and were used in both the algorithms, 'bd-cell' and 'bd-fast'.

1. Including $\{0, 0, 0\}$ in box-neighbors and removing the while loop for calculation of within box interactions
2. Adding a ϵ to prevent divide by zero condition
3. Including the forces calculations inside the interaction function and thereby getting rid of pairs and distances arrays
4. Fixing Boxdim = $(L/4) * a$ (where a is the radius of the molecule), so that we divide the cells into as small as possible to just fit minimum number of particles was an important factor contributing to considerable speed up
5. Replace the idx, idy and idz by index, so that we can parallelize the loop to calculate forces for particles in different boxes simultaneously

2.4. Difference between 'bd-cell' and 'bd-fast' algorithms

The cell updation (assigning particles to cells) part in 'bd-cell' is serial while in 'bd-fast' it is parallelized. We can put a 'critical' pragma around the part of updating the 'next' list and '$b \rightarrow head$' to prevent the race condition, but this becomes a bottleneck to parallelize the cell updation part as 'critical' pragma is very slow.

One very interesting observation is, let us calculate the probability that the above race condition occurs while updating cell lists.

Say, we have a Brownian Dynamics simulation problem with P = number of particles, N_t = number of threads, L = box width.

Let us look at what is the probability of 2 or more threads writing to the same box at the same time? While parallelizing the for loop using # pragma omp for, if we take static scheduling for N_t threads, we get P/N_t different sections of the 'for' loop.

Now, calculating the probability of particles from one box occurring in such 2 or more sections simultaneously is:

Probability for race condition = $({}^{N_t}C_2) * ((\frac{4}{L})^3 * N_t)^2 * (\frac{N_t}{P})^2 + \ldots + ({}^{N_t}C_k) * ((\frac{4}{L})^3 * N_t)^k * (\frac{N_t}{P})^k + \ldots + ({}^{N_t}C_{N_t}) * ((\frac{4}{L})^3 * N_t)^{N_t} * (\frac{N_t}{P})^{N_t}$

In the case of $P = 10000$, say running of $N_t = 100$ threads with box width $L = 60$, the probability for getting a race condition is (Note: the 1^{st} term is significant compared to other terms), so just calculating the first term of the above equation, we get

Probability for race condition in one time step = $({}^{100}C_2) * ((\frac{4}{60})^3 * 100)^2 * (\frac{100}{10000})^2$ which approximately is $4e-05$ and can be neglected.

Hence, 'bd-fast.c' can be safely used for simulations where particles P are very large in number and also the density is less. (as vol. fraction ϕ is inversely proportional to L). The empirical results demonstrate the same.

Table 2. Comparison of Best Timings under various settings with parallelization done using OpenMP. Number of threads = 240

Make [option]	SIMD flag	OpenMP flag	Runtime (secs)	Speedup
mic-novec	0	0	19.317257	1.0
only-openmp	0	1	0.181829	106.2384
only-vec	1	0	0.832667	23.1992
all-on	1	1	0.008654	2232.2488

Table 3. Comparison of Best Timings under various settings with parallelization done using Cilk Plus. Number of threads = 240

Make [option]	SIMD flag	Cilk flag	Runtime (secs)	Speedup
mic-novec	0	0	19.317257	1.0
only-cilk	0	1	0.212126	91.0650
only-vec	1	0	0.86272	22.3911
all-on	1	1	0.01526	1265.8768

3. Hydrodynamic Interactions

The project2 framework was used for this part. I was able to successfully vectorize the complete code and have attached the vectorization report in git repository. Also, the code is parallelized using OpenMP and Cilk Plus, comparison of which is shown in the following sections.

3.1. Correctness of implementation

I verified the corrections of my implementation by checking that all simulations give zero squared error. Simulations were done on 'mic2' coprocessor and results were obtained using the input file 'lac1-novl2.xyz' and the values of parameters $x_i = 1.5\pi/L$, $n_r = 2$ and $n_k = 3$ were fixed throughout.

3.2. Overall Speed Comparison

Let us define 'Vectorization': SIMD (off=0, on=1) and 'Multithreading': OpenMP/Cilk (off=0, on=1). Tables[2 and 3] are the overall runtimes and speedup obtained by using OpenMP and Cilk Plus respectively. We can observe that the runtimes obtained from OpenMP are faster than Cilk Plus.

The figures[3, 4] compare the 'average time per iteration', with and without vectorization, between OpenMP and Cilk Plus.

3.3. Speedup graphs

The speedup is (runtime-for-novec-noparallelization) / (current-runtime) and the comparison between OpenMP and Cilk Plus are shown in the figures[5 , 6]. I obtained almost 24x speed up due to vectorization alone. An important observation is somehow the performance of Cilk with

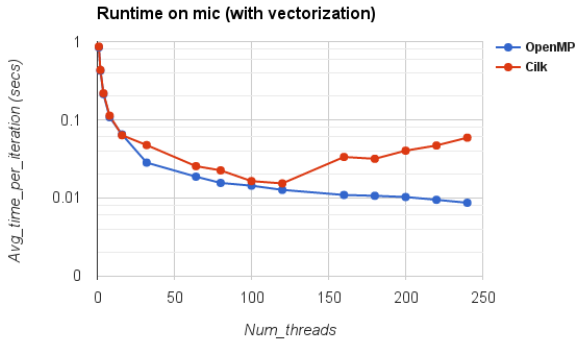

Figure 3. Log plot of Runtime vs Number of threads (with vectorization). Comparison between OpenMP vs Cilk Plus versions.

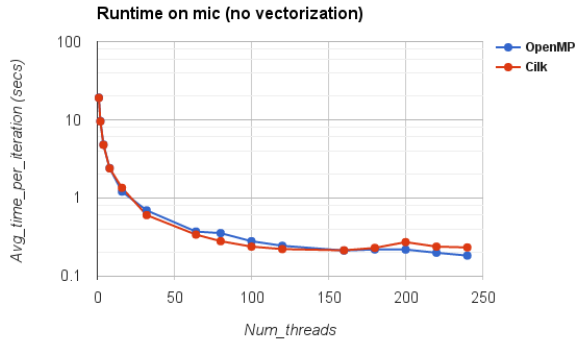

Figure 4. Log plot of Runtime vs Number of threads (without vectorization). Comparison between OpenMP vs Cilk Plus versions.

vectorization is not at par with the OpenMP version, the reasons of which should be further explored.

3.4. Optimizations done

Different types of parameters tried:

1. KMP_AFFINITY=scatter/balanced with granularity=fine were the best choices. KMP_AFFINITY= compact did not give good results as expected
2. For OpenMP implementation, tried various OMP-SCHEDULE = dynamic, static, guided with various choices, but the best results were obtained by static scheduling. I guess, since the number of particles are less, the for loops are not that large for other scheduling techniques to exploit their optimum functioning
3. Various NUMA options tried for mic

Optimizations done in code:

1. In Real space sum calculation loop (for j=i changed to j=i+1) and the [if(i==j) and if(r2==0)] conditions were removed

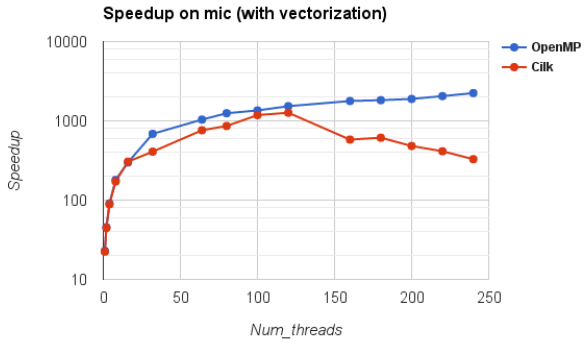

Figure 5. Log plot of Runtime vs Number of threads (with vectorization). Comparison between OpenMP vs Cilk Plus versions.

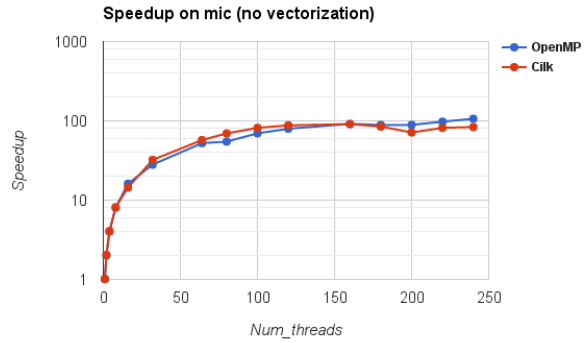

Figure 6. Log plot of Runtime vs Number of threads (without vectorization). Comparison between OpenMP vs Cilk Plus versions.

2. The if (i==j) condition in reciprocal calculations were also removed and was taken care in the 'self-part' section while updating array 'a'
3. The 3 for loops going from $-n_k/n_r$ to n_k/n_r were replaced by indices (requires only a single for loop) using which x,y,z were recalculated
4. Functions were made inline and restricts were used
5. The [goto out] condition was also removed by iterating to half the size of original loop
6. The remaining 2 for loops were flattened to obtain improved performance while running openmp pragmas or Cilk (Some calculations were tricky to flatten the for loops, which can be observed from the code itself)
7. Even tried flattening the complete for loops in real part calculations by adjusting the variables and removing the 'temp' array and directly updating 'a' instead, but the overhead of accessing 'a' was more and the runtime increased
8. The temp and other arrays, array size was adjusted to avoid cache sharing
9. Further improvements can be done to avoid possible false sharing for accessing the array 'a' by the threads

4. Cholesky Decomposition

Once the hydrodynamic interactions are calculated, it returns a full symmetric matrix A of dimension $(3N$ by $3N)$, where N is number of particles. Now, we need to decompose the matrix $A = L.L^T$ into product of 2 lower triangular matrices using Cholesky decomposition. Then, we multiply this lower triangular matrix L with a random Gaussian vector and together with steric forces, we update the particle positions in every iteration.

4.1. Implementation

I have 3 versions for calculating Cholesky decomposition:

1. Using the Math Kernel Library (MKL) in-built functions like LAPACKE_dpotrf, cblas_dgemm etc. These are parallel and vectorized implementations optimized for Xeon Phi Coprocessor.
2. My vectorized and OpenMP/Cilk versions inspired by RosettaCode
3. Using the standard MATLAB functions

4.2. Implementation details and runtime comparisons

Figure7 compares run times for Cholesky decomposition part with input file as 'lac1-novl2.xyz' having $N = 257$ particles with varying number of threads and vectorization enabled. Cholesky decomposition was done using the in-built MKL functions.

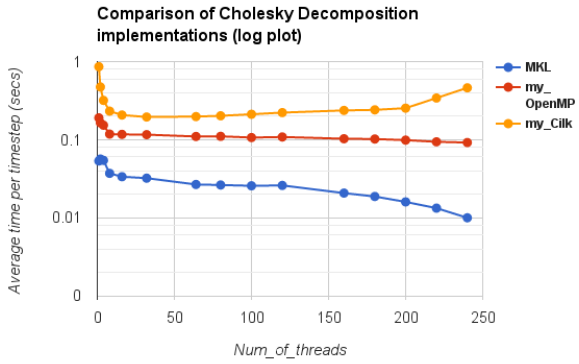

Figure 7. Log plot of Runtime vs Number of threads. Comparison between OpenMP vs Cilk Plus vs MKL versions.

We can see that the MKL version is faster as expected. In my versions, I was able to rearrange the original rosetta code such that it can be parallelized. Though, I was not able to exploit the vectorization part that well. This may be one of the major reasons why my implementation is slower than

Table 4. Profiling the code. Runtime for various parts are calculated for optimum vectorization and parallelization settings. The simulation was run for 1000 intervals with NThreads = 200, N = 257 particles and average runtime for a single interval is reported. The Steric Interactions and Hydrodynamics part have comparable runtimes in scalable settings.

Different parts	OpenMP Runtime (secs)	Cilk Runtime (secs)
Steric Interactions	0.02410	0.02824
Hydrodynamic Interactions	0.01070	0.15836
Cholesky Decomposition	0.00999	0.04699
Updating Position	0.00465	0.04805

the state-of-the-art MKL version. Anyway, the OpenMP version of my implementation is faster than that of Cilk Plus as evident from Figure7

5. Overall Brownian Dynamics analysis

In this section, we will first go through the profiling of the complete Brownian Dynamics code and identify the bottlenecks. The following section will show the runtime and speedup graphs to simulate the complete BD.

5.1. Instructions to run the code

For OpenMP version: 1.make bd_omp 2.make runmic

For OpenMP version with my implementation of Cholesky: 1.make bd_omp_myc 2.make runmic

For Cilk: 1.make bd_cilk 2.make runcilk

For Cilk version with my implementation of Cholesky: 1.make bd_cilk_myc 2.make runcilk

We need to set the number of threads and other parameters like KMP_AFFINITY etc. manually in either makefile or in the bd code.

5.2. Profiling the BD code

Table4 shows the approximate runtimes for different parts. As previously described, the BD code can be roughly divided into 4 different parts which comprise steric interactions, hydrodynamic interactions, cholesky decomposition and multiplying lower triangular matrix L with random gaussian vector and updating the positions. The Steric Interactions and Hydrodynamics part have comparable runtimes in scalable settings and further optimizations should be focused on improving these parts.

5.3. Runtime Comparisons

Figure9 shows the runtime comparisons under optimum settings between OpenMP and Cilk Plus versions. Note, for Cilk implementation, I also change the grain size in runtime to achieve better results.

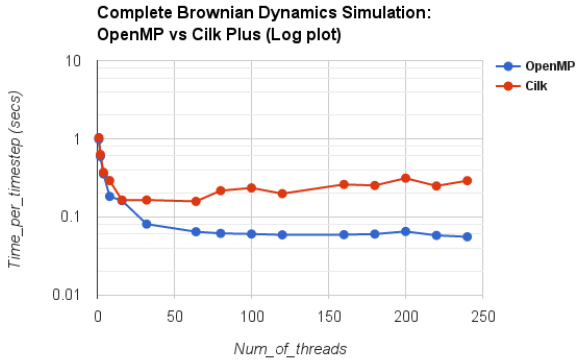

Figure 8. Log plot of Runtime vs Number of threads. Comparison between OpenMP vs Cilk Plus for complete BD with optimum settings.

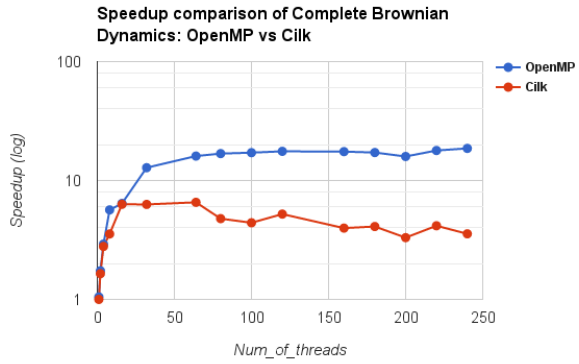

Figure 9. Log plot of Speedup vs Number of threads. Comparison between OpenMP vs Cilk Plus for complete BD with optimum settings.

6. Ease of Programming

Cilk Plus is definitely easier to code with its simpler syntax and less number of parameters to consider. OpenMP has many scheduling options and also we have to keep track of shared and private variables. One issue with using for (cilk_for) in Cilk Plus is that it considers all the variables declared outside the scope of for loop as shared variables. So, if anyone follows the old style of C coding, where we initialise all the variables at the start, this can be a important point to remember as it can result in considerable slowdown if private variables are declared outside the scope of cilk_for. Sometimes, handling the grain parameter of Cilk can be tricky and may lead to significant slow down if not set properly. In my opinion, the documentation of OpenMP available online is way more articulate and well updated than Cilk Plus. As both of these platforms follow different

principals for parallelization and given the ease of implementation, it doesn't hurt to check performance using both for a given task.

7. Conclusions and Acknowledgments

I implemented the complete Brownian Dynamics code. Analysed various parts of the code and identified bottlenecks. Did extensive simulations to compare between OpenMP and Cilk Plus platforms. Also, vectorized the code to get further speedup. In my simulations, I found that runtimes are better for OpenMP as compared to Cilk, but Cilk is definitely easier to code. 3D simulation was done using the VMD tool for 257 particles and the output file is saved in git. Would be interesting to combine this code with MPI to run for more number of particles.

Finally, I would like to take this opportunity to thank Prof. Edmond Chow for his articulate explanation of HPPC concepts. Also, hearty thanks to Stephen (TA), Patrick and Matt for all our productive discussions.