Noname manuscript No.
(will be inserted by the editor)

Parallel simulation of Brownian dynamics on shared memory systems with OpenMP and Unified Parallel C

Carlos Teijeiro · Godehard Sutmann · Guillermo L. Taboada · Juan Touriño

Received: date / Accepted: date

Abstract The simulation of particle dynamics is an essential method to analyze and predict the behavior of molecules in a given medium. This work presents the design and implementation of a parallel simulation of Brownian dynamics with hydrodynamic interactions for shared memory systems using two approaches: (1) OpenMP directives and (2) the Partitioned Global Address Space (PGAS) paradigm with the Unified Parallel C (UPC) language. The structure of the code is described, and different techniques for work distribution are analyzed in terms of efficiency, in order to select the most suitable strategy for each part of the simulation. Additionally, performance results have been collected from two representative NUMA systems, and they are studied and compared against the original sequential code.

Keywords Brownian dynamics · parallel simulation · OpenMP · PGAS · UPC · shared memory

1 Introduction

Particle based simulation methods have been continuously used in physics and biology to model the behavior of different elements (e.g., molecules, cells) in a medium (e.g., fluid, gas) under thermodynamical conditions (e.g., temperature, density). These methods represent a simplification of the real-world

C. Teijeiro, G. L. Taboada, J. Touriño
Computer Architecture Group, University of A Coruña
15071 A Coruña, Spain
E-mail: {cteijeiro,taboada,juan}@udc.es

G. Sutmann
Jülich Supercomputing Centre (JSC), Institute for Advanced Simulation
Forschungszentrum Jülich GmbH
52425 Jülich, Germany
ICAMS, Ruhr-University Bochum, D-44801 Bochum, Germany
E-mail: g.sutmann@fz-juelich.de

scenario but often provide enough details to model and predict the state and evolution of a system on a given time and length scale. Among them, Brownian dynamics describes the movement of particles in solution on a diffusive time scale, where the solvent particles are taken into account only by their statistical properties, i.e. the interactions between solute and solvent are modeled as a stochastic process.

The main goal of this work is to provide a clear and efficient parallelization for the Brownian dynamics simulation, using two different approaches: OpenMP and the PGAS paradigm. On the one hand, OpenMP facilitates the parallelization of codes by simply introducing directives (pragmas) to create parallel regions in the code (with private and shared variables) and distribute their associated workload between threads. Here the access to shared variables is concurrent for all threads, thus the programmer should control possible data dependencies and deadlocks. On the other hand, PGAS is an emerging paradigm that treats memory as a global address space divided in private and shared areas. The shared area is logically partitioned in chunks with affinity to different threads. Using the UPC language [15] (which extends ANSI C including PGAS support) the programmer can deal directly with workload distribution and data storage by means of different constructs, such as assignments to shared variables, collective functions or parallel loop definitions.

The rest of the paper presents the design and implementation of the parallel code. First, some related work on this area is presented, and then a mathematical and computational description of the different parts in the sequential code is provided. Next, the most relevant design decisions taken for the parallel implementation of the code, according to the nature of the problem, are discussed. Finally, a performance analysis of the parallel code is accomplished, and the main conclusions are drawn.

2 Related work

There are multiple works on simulations based on Brownian dynamics, e.g. several simulation tools such as BrownDye [7] and the BROWNFLEX program included in the SIMUFLEX suite [5], and many other studies oriented to specific fields, such as DNA research [8] or copolymers [10]. Although most of these works focus on sequential codes, some of them have developed parallel implementations with OpenMP, applied to smoothed particle hydrodynamics (SPH) [6] and the transportation of particles in a fluid [16]. The UPC language has also been used in a few works related to particle simulations, such as the implementation of the particle-in-cell algorithm [12]. However, comparative analyses between parallel programming approaches on shared memory with OpenMP and UPC are mainly restricted to some computational kernels for different purposes [11,17], without considering large simulation applications.

The most relevant recent work on parallel Brownian dynamics simulation is BD_BOX [2], which supports simulations on CPU using codes written with MPI and OpenMP, and also on GPU with CUDA. However, there is still little

information published about the actual algorithmic implementation of these simulations, and their performance has not been thoroughly studied, especially under periodic boundary conditions. In order to solve this and provide an efficient parallel solution on different NUMA shared memory systems, we have analyzed the Brownian dynamics simulation for different problem sizes, and described how to address the main performance issues of the code using OpenMP and a PGAS-based approach with UPC.

3 Theoretical description of the simulation

The present work focuses on the simulation of Brownian motion of particles including hydrodynamic interactions. The time evolution in configuration space has been stated by Ermak and McCammon [3] (based on the Focker-Planck and Langevin descriptions) which includes the direct interactions between particles via systematic forces as well as solvent mediated effects via a correlated random process

$$\mathbf{r}_i(t+\Delta t) = \mathbf{r}_i(t) + \sum_j \frac{\partial \mathbf{D}_{ij}(t)}{\partial \mathbf{r}_j}\Delta t + \sum_j \frac{1}{k_B T}\mathbf{D}_{ij}(t)\mathbf{F}_j(t)\Delta t + \mathbf{R}_i(t+\Delta t) \quad (1)$$

Here the underlying stochastic process can be discretized in different ways, leading to e.g. the Milstein [9] or Runge-Kutta [14] schemes, but in Eq. 1 the time integration method follows a simple Euler scheme. In a system that contains N particles, the trajectory $\{\mathbf{r}_i(t); t \in [0, t_{max}]\}$ of particle i is calculated as a succession of small and fixed time step increments Δt, defined as the sum of interactions on the particles during each time step (that is, the partial derivative of the initial diffusion tensor $\mathbf{D}^0_{ij}$ with respect to the position component and the product of the forces $\mathbf{F}$ and the diffusion tensor at the beginning of a time step, being k_B Boltzmann's constant and T the absolute temperature), and a random displacement $\mathbf{R}_i(\Delta t)$ associated to the possible correlations between displacements for each particle. The random displacement vector $\mathbf{R}$ is Gaussian distributed with average value $\langle \mathbf{R}_i \rangle = 0$ and covariance $\langle \mathbf{R}_i(t+\Delta t)\mathbf{R}_j^T(t+\Delta t)\rangle = 2\mathbf{D}_{ij}(t)\Delta t$. The diffusion tensor $\mathbf{D}$ thereby contains information about hydrodynamic interactions, which, formally, can be neglected by considering only the diagonal components $\mathbf{D}_{ii}$. Choosing appropriate models for the diffusion tensor, the partial derivative of $\mathbf{D}$ drops out of Eq. 1, which is true for the Rotne-Prager tensor [13] used in this work. Therefore, Eq. 1 reduces to

$$\Delta \mathbf{r} = \frac{1}{kT}\mathbf{D}\mathbf{F}\Delta t + \sqrt{2\Delta t}\,\mathbf{S}\,\boldsymbol{\xi} \quad (2)$$

where the expression $\mathbf{R} = \sqrt{2\Delta t}\mathbf{S}\boldsymbol{\xi}$ holds and $\mathbf{D} = \mathbf{S}\mathbf{S}^T$, which relates the stochastic process to the diffusion matrix. Therefore, $\mathbf{S}$ may be calculated via a Cholesky decomposition of $\mathbf{D}$ or via the square root of $\mathbf{D}$. Both approaches are very CPU time consuming and have a complexity of $\mathcal{O}(N^3)$. A faster approach

that approximates the random displacement vector $\mathbf{R}$ without constructing $\mathbf{S}$ explicitly was introduced by Fixman [4] using an expansion of $\mathbf{R}$ in terms of Chebyshev polynomials, which has a complexity of $\mathcal{O}(N^{2.25})$.

The interaction model for the computation of forces in the system (which here considers periodic boundary conditions) consists of (1) direct systematic interactions, which are computed as Lennard-Jones type interactions and evaluated within a fixed cutoff radius, and (2) hydrodynamic interactions, which have long range character. The latter ones are reflected in the off-diagonal components of the diffusion tensor $\mathbf{D}$ and therefore contribute to both terms of the right-hand side of Eq. 2. Due to the long range nature, the diffusion tensor is calculated via the Ewald summation method [1].

4 Design of the parallel simulation

The sequential code considers a unity 3-dimensional box with periodic boundary conditions that contains a set of N equal particles. The main part of this code is a `for` loop in which each iteration represents a time step. Following the formulas presented in Section 3, each time step of the simulation consists of two main parts: (1) the calculation of the forces that act on each particle (function `calc_force`), and (2) the computation of random displacements for every particle in the system (function `covar`).

The main loop of time steps is not parallelizable, as each iteration depends on the previous one. However, the workload of each iteration can be executed in parallel through a domain decomposition of the work. Here OpenMP requires separate parallel regions for each function, as a work distribution is only possible if the parallel region is defined inside the target function; otherwise, its arguments would be treated mandatorily as private variables. Nevertheless, UPC targets a balanced workload distribution between threads and an efficient data access, exploiting the NUMA architecture through the maximization of data locality (processing the data stored in the thread's affine space). A more detailed description of both OpenMP and UPC parallelizations is given in the next subsections.

4.1 Calculation of interactions

The main part of the computation in function `calc_force` consists in updating the diffusion tensor $\mathbf{D}$, which is defined in the code as a symmetric square `pNDIM`×`pNDIM` matrix (called `D`), where `pNDIM`$= 3 \times N$ (that is, one value per dimension and per pair of particles). For each element in matrix `D`, the Ewald summation is split into (1) a short range part, where elements are calculated according to pairwise distances within a cutoff radius, and (2) a long range part, which is evaluated in Fourier k-space. Both computations are performed individually for each value in matrix `D` using the current coordinates for each particle in the system with two separate loops. In the sequential code, these loops operate only on the upper triangular part of `D`, because of its symmetry.

According to the previous definitions, the independent computation of values of matrix `D` defines a straightforward parallelization for `calc_force` in the simulation. The OpenMP implementation uses an `omp for` directive to parallelize the computation of short and long range interactions, using a dynamic scheduling of iterations because of the different amount of values calculated per particle/dimension, alongside with `atomic` directives to obtain the average overall speed and pressure values. The UPC code performs an analogous processing, but using a 1D storage for matrix `D`: its elements are divided in `THREADS` equal chunks (being `THREADS` the number of threads in the UPC program), and each thread computes its associated values for each particle for all dimensions.

4.2 Computation of random displacements

The random displacements are calculated for each particle as Gaussian random numbers with prescribed covariances according to the diffusion tensor matrix. The simulation code includes two alternative methods to obtain the covariance terms: (1) a Cholesky decomposition of `D` and (2) the Fixman's approximation method.

The Cholesky decomposition of matrix `D` obtains a lower triangular matrix `L`, whose values for each row are multiplied by a random value generated for the corresponding particle and dimension. The final result is the sum of all these products, as stated in Eq. 3, where row `i` represents particle `i` *div* `DIM` in dimension `i` *mod* `DIM` (with `DIM`= 3, the number of dimensions), and `xic` is the array of `pNDIM` random values:

$$\mathbf{R}_i(t + \Delta t) = \sum_{j=1}^{i} \texttt{L[i][j]} \cdot \texttt{xic[j]} \tag{3}$$

The original sequential algorithm calculates the first three values of the lower triangular result matrix (that is, the elements of the first two rows), and then the remaining values are computed row by row in a loop. Here the data dependencies between the values in `L` computed in each iteration and the ones computed in previous iterations prevent the direct parallelization of the loop. However, UPC takes advantage of its data manipulation facilities and computes the values in `L` as a sum (reduction) of different contributions calculated by each thread: after a row in matrix `L` is computed, its elements are used to obtain partial computations of the random displacements associated to the unprocessed rows, thus maximizing parallelism. This functionality is implemented in UPC using the `upc_forall` construct to distribute the loop iterations between threads, and a broadcast collective to send partial computations. Listing 1 presents a pseudocode of this algorithm.

Regarding OpenMP, and given the difficulties for an efficient parallelization of the original algorithm commented previously, an alternative iterative algorithm is proposed: the first column of the result matrix is calculated by

```
for every row 'i' in matrix D
  if the row is assigned to MYTHREAD
    obtain values of row 'i' in L
    calculate random displacement 'i'
  endif

  broadcast values in row 'i' to all threads

  upc_forall rows 'j' located after 'i' in matrix D
    calculate partial values for elements in row 'j'
    calculate partial value for displacement 'j'
  endfor
endfor
```

Listing 1 UPC pseudocode for Cholesky decomposition

dividing the values of the first row in the source matrix by its diagonal value (parallelizable `for` loop), then these values are used to compute a partial contribution to the rest of elements in the matrix (also parallelizable `for` loop). Finally, these two steps are repeated to obtain the rest of columns in the result matrix. Listing 2 shows the source code of the iterative algorithm: a `static` scheduling is used in the first two loops, whereas the last two use a `dynamic` one because of the different workload associated to their iterations.

```
 L[0][0] = sqrt(D[0][0]);
#pragma omp parallel private(i,j,k)
{
#pragma omp for schedule(static)
 for (j=1; j<N*DIM; j++)
   L[j][0] = D[0][j]/L[0][0];
#pragma omp for schedule(static)
 for (j=1; j<N*DIM; j++)
   for (k=1; k<=j; k++)
     L[j][k] = D[k][j]-L[0][j]*L[0][k];

 for (i=1; i<N*DIM; i++) {
   L[i][i] = sqrt(L[i][i]);
#pragma omp for schedule(dynamic)
   for (j=i+1; j<N*DIM; j++)
     L[j][i] = L[i][j]/L[i][i];

#pragma omp for schedule(dynamic)
   for (j=i+1; j<N*DIM; j++)
     for (k=i+1; k<=j; k++)
       L[j][k] -= L[i][j]*L[i][k];
 }
}
```

Listing 2 Iterative Cholesky decomposition with OpenMP

Fixman's algorithm [4] is an alternative to Cholesky decomposition to obtain the covariance terms of random displacements. This method approximates $\mathbf{R}$ using the Chebyshev polynomial $\mathbf{S_M\xi}$, where $\mathbf{S}_M$ is an approximation to

S (cf. Eq. 2) and the number of polynomial coefficients M controls the accuracy of the approximation, without constructing an additional matrix as with Cholesky decomposition. Practically this algorithm works on a scaled matrix, with eigenvalues within the range $\lambda \in [-1, 1]$, thus requiring to know the smallest (λ_{min}) and largest (λ_{max}) eigenvalues of matrix **D**, as stated in Eq. 4.

$$\begin{aligned} &\sqrt{d} \approx \sum_{l=0}^{L} a_l C_l \text{ , where } L \equiv \text{order of the polynomial approximation, and} \\ &C_0 = 1 \;\; ; \;\; C_1 = d_a d + d_b \;\; ; \;\; C_{l+1} = 2(d_a d + d_b) C_l - C_{l-1} \text{ , having} \\ &d_a = \frac{2}{\lambda_{max} - \lambda_{min}} \;\; ; \;\; d_b = \frac{\lambda_{max} + \lambda_{min}}{\lambda_{max} - \lambda_{min}} \end{aligned} \tag{4}$$

According to this, two iterative approximation methods are used for Fixman's algorithm: (1) the calculation of the minimum and maximum eigenvalues of matrix `D`, which uses a variant of the power method, and (2) the computation of covariance terms following the formulas in Eq. 4. In both cases, the core of the algorithm consists of two `for` loops that operate on every row of matrix `D` independently (thus fully parallelizable) until the predefined accuracy is reached. Each iteration of these methods requires all the approximations computed for every particle in all dimensions (`pNDIM` values) in the previous iteration, thus both codes use two arrays that are read or written alternatively on each iteration to avoid additional data copies.

The OpenMP code includes `omp for` directives for each loop to compute approximations, using a `critical` directive to obtain the maximum/minimum eigenvalues of `D` and a `reduction` clause to compute the total error value for the final approximated covariance terms. Listing 3 presents the pseudocode of the loop that computes the maximum eigenvalue of `D`, using `x` and `x_d` as working arrays for the iterative method. Here a `reduction` clause with a maximum operator could be used instead of the `critical` directive, but in this case the experimental analysis showed better performance using `critical`.

The UPC parallelization requires the use of three collective operations (all-to-all, allgather and allreduce) to obtain the final results of both iterative methods. A prototypical implementation of their associated loops is shown in Listing 4, where the approximations are stored alternatively in arrays `x` and `x_d`, as in the code in Listing 3. Each thread computes a partial result for the approximated eigenvalues in its local memory, and then an all-to-all collective communication is performed to get all the partial results of their assigned particles. After that, each thread obtains an approximation of its associated maximum eigenvalue, and finally an allgather collective ensures that all threads have all the approximated values in order to prepare a new iteration of the approximation method. At the end of the `while` loop, the maximum eigenvalue is obtained with a reduction of the partial values computed on each thread.

In general terms, the presented UPC parallelization required additional lines of source code (SLOCs) than using OpenMP. This is due to the data

```
#pragma omp parallel
{
while the required accuracy for 'eigmax' is not reached
do
  if the iteration number is even
    // read from 'x', write to 'x_d'
#pragma omp for schedule(static) \
    default(shared) private(i,j,...)
    for every particle 'i' in the system
      x_d[i] = 0;
      for every particle 'j' in the system
        x_d[i] += D[i][j]*x[j]/eigmax;
      endfor
#pragma omp critical
      if it is the maximum value
        select it in 'eigmax'
      endif
    endfor
  else
    // The same code as above, but changing
    // 'x' with 'x_d' and vice versa
  endif
endwhile
}
```

Listing 3 OpenMP pseudocode that computes the maximum eigenvalue of matrix `D`

movements, especially collective operations, that are needed for an efficient access to the data set (logically stored in the thread's shared and private memory spaces). However, thanks to these data movement operations, the UPC code can take advantage of efficient data layouts both for NUMA and distributed memory systems [11]. This higher flexibility of UPC contrasts with the limitation of UPC collectives, which can only be used on 1D arrays and thus the code was adapted accordingly, as mentioned at the end of Section 4.1. Regarding OpenMP, the parallelization was mainly accomplished by introducing directives in the source code, but this code also required major changes in order to obtain scalability (as it was the case for Cholesky decomposition).

5 Performance evaluation

Two different NUMA systems have been used for the performance analysis. The first one (named NEH in the graphs) consists of 2 Intel Xeon X5570 (Nehalem-EP) quad-core processors at 2.93 GHz with Simultaneous Multi-threading (SMT) and 24 GB of memory at 1066 MHz. The second system (MGC) is an HP ProLiant SL165z G7 node with 2 dodeca-core AMD Opteron processors 6174 (Magny-Cours) at 2.2 GHz with 32 GB of memory. The Intel C Compiler (icc) v12.1 and the Open64 Compiler Suite (opencc) v4.2.5.2 have been used as OpenMP compilers in NEH and MGC, respectively. The UPC compiler in both systems is Berkeley UPC 2.14.2 (a UPC-to-C source-to-source compiler) with the SMP conduit, backed by the aforementioned C compilers

```
while the required accuracy for 'eigmax' is not reached
do
  if the iteration number is even
    // read from 'x', write to 'x_d'
    for every element in 'D' (associated to each thread)
      compute partial approximation in 'copyArray' using 'x'
    endfor
    all-to-all collective to get all partial \
     approximations from 'copyArray'
    for every  particle 'i' in the system
      for every 'THREADS' partial results in 'copyArray'
        sum partial results to get the final \
         approximation of 'x_d[i]'
      endfor
      if it is the maximum value
        select it in 'eigmax'
      endif
    endfor
    gather all the computed values in 'x_d'
  else
    // The same code as above, but changing
    // 'x' with 'x_d' and vice versa
  endif
  reduce all the 'eigmax' values on each thread
endwhile
```

Listing 4 UPC pseudocode that computes the maximum eigenvalue of matrix `D`

on each system. The optimization flags used in all executions are `-fast` for icc and `-Ofast` for opencc. The environment variable `OMP_STACKSIZE` has been set to a small value (128 KB) for OpenMP executions in order to obtain the highest efficiency. Periodic boundary conditions (3×3 boxes per dimension) are considered for all simulations. Next some relevant performance results of specific parts of the code (Tables 1-3) are detailed, and after them Figures 1 and 2 show data of the whole simulation.

Table 1 Execution time profile of the sequential code

Code Part	128 particles	1024 particles
`calc_force` - short range contributions	1.104 s	70.125 s
`calc_force` - long range contributions	2.954 s	522.867 s
`covar` - option 1: Cholesky	0.838 s	435.748 s
`covar` - option 2: Fixman	0.265 s	28.327 s
`move`	0.006 s	0.483 s
Total time (with Cholesky)	4.902 s	1029.223 s
Total time (with Fixman)	4.329 s	621.802 s

Table 1 shows the execution time breakdown of sequential simulations (128 and 1024 particles with 100 time steps) in the NEH system, considering the functions commented in Section 4 and including the computation of the new coordinates for the next time step as a separate function (called `move`). These

results give a measure of the different weights and algorithmic complexities of each part of the code. As the diffusion tensor matrix `D` has $(3 \times N)^2$ elements, its construction has a complexity of at least $\mathcal{O}(N^2)$, which is related to the computation of short and long range interactions. However, the long range part has an additional contribution because of the use of reciprocal vectors to obtain the required error tolerance in the Ewald sum, which increases its complexity to approximately $\mathcal{O}(N^{2.5})$. Regarding the computation of random displacements (function `covar`), the results clearly show the effect of the higher complexity of the Cholesky decomposition with respect to Fixman's algorithm.

Table 2 presents performance results of the loop that computes short range interactions (see Section 4.1) in a 1024-particle system with 100 time steps using the NEH system, with initial conditions chosen as a regular grid. These results confirm that the OpenMP `dynamic` distribution of iterations outperforms the `static` ones (block and cyclic) because of the different workload associated to each iteration, obtaining about 7.56 of speedup with 8 threads. This speedup is limited by the sequential calculations of pressure and speed using the `atomic` directive, which creates a lightweight critical section that solves the data dependencies. The UPC version obtains similar results (slightly better for 8 threads) to the most efficient OpenMP implementation.

Table 2 Execution times of short range interactions with OpenMP and UPC

Short range interactions - 1024 particles - 100 time steps				
#Threads	**OMP-static block**	**OMP-static cyclic**	**OMP-dyn.**	**UPC**
1	70.125 s			
2	52.628 s	37.827 s	34.836 s	34.876 s
4	31.195 s	20.567 s	17.143 s	17.123 s
8	17.871 s	14.545 s	9.270 s	8.876 s

Table 3 presents the results of OpenMP and UPC Cholesky decomposition algorithms (see Section 4.2) using 1024 particles with 100 time steps on NEH, which shows that the original implementation (used in UPC) is the most efficient sequential routine. Even though the OpenMP iterative version obtains scalability up to 8 threads, it is far from the ideal because of loop `i` presented in Listing 2: implicit synchronizations executed at the end of each nested loop `j` due to the `omp for` directive, thus involving a significant overhead.

Table 3 Execution times of `covar` using Cholesky with 1024 particles and 100 time steps

Code \ #Threads	**1**	**2**	**4**	**8**
OpenMP	568.153 s	396.564 s	354.792 s	348.334 s
UPC	435.748 s	345.924 s	250.175 s	182.432 s

Figure 1 presents the execution times and speedups of the whole simulation with 128 particles and 100 time steps. The execution times of NEH are better than those of MGC for both random displacement generation methods, even though the influence of using SMT for 16 threads affects performance. This is mainly due to the higher processor power of NEH. The reduced problem size involves a low average workload per thread, thus the bottlenecks of the code (e.g., OpenMP `atomic` directives and UPC collective communications) are more noticeable. Nevertheless, the UPC implementation compiled with Berkeley UPC even obtains superlinear speedup for 2 and 4 threads with Fixman on NEH. Regarding the `covar` function, the use of Fixman's method clearly helps to obtain higher performance when compared to Cholesky decomposition, as stated in the previous sections.

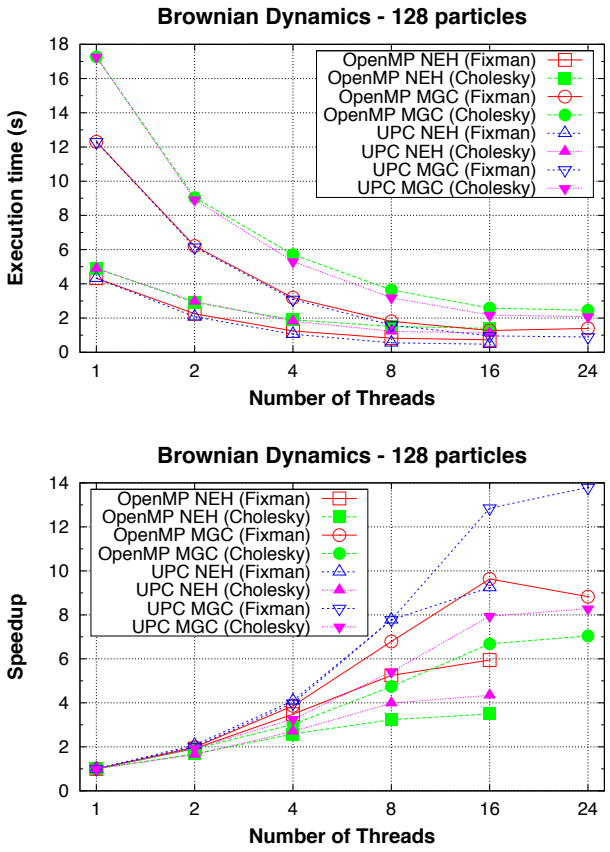

Figure 1 Performance results with 128 particles (Fixman and Cholesky)

Figure 2 shows the simulation results for a large number of particles (4096) and 5 time steps. Only the results of `covar` with Fixman's algorithm are shown, because of its higher performance. The results up to 8 threads present similar performance, which is mainly due to the heavy computation of long range interactions compared to the relatively small computation of random displacements. In fact, the larger problem size provides higher speedups than those of Figure 1. However, the algorithmic complexity of calculating the diffusion tensor $\mathbf{D}$ is $\mathcal{O}(N^2)$, whereas Fixman's algorithm is $\mathcal{O}(N^{2.25})$; thus, when the problem size increases, the generation of random displacements represents a larger percentage of the total simulation time. As a result of this, and also given the parallelization issues commented in Section 4.2, the speedup is slightly limited for 16 or more threads, mainly for OpenMP (also due to the use of SMT in NEH). However, considering the distance to the ideal speedup, we can conclude that both systems present reasonably good speedups for this code.

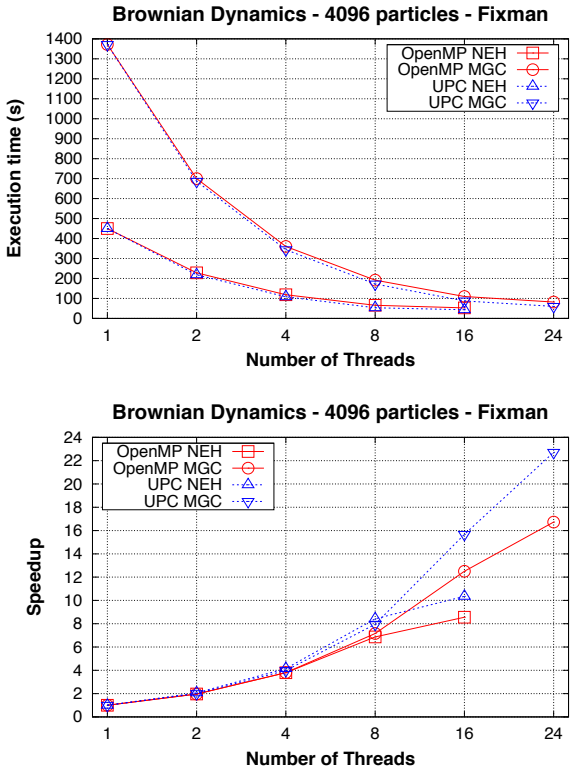

Figure 2 Performance results with 4096 particles (Fixman)

6 Conclusions

This work has presented the parallel implementation of a Brownian dynamics simulation code on shared memory using OpenMP directives and a PGAS-based approach with the UPC language. According to both programming paradigms, different workload distributions have been presented and tested for the main functions in the code, trying to obtain good performance as the number of threads increases. The scheduling techniques in `for` loops, the convenient use of `reduction` or `critical` directives, and the control of the stack size in parallel function calls have been the most relevant features to tune the OpenMP code, whereas the efficient work distribution and a wise optimization of data management have been key factors for UPC. The results of the parallel implementation have shown that there is a high dependency of the code on the part where random displacements are generated for each particle, and here Fixman's approximation method has shown to be the best choice. Regarding both programming paradigms, UPC has obtained better results than OpenMP, mainly because of the more efficient access to the data set, as the PGAS memory model presents a more straightforward adaptation to current NUMA multicore-based systems, although at the cost of a slightly higher programming effort. OpenMP has provided a more compact implementation in terms of SLOCs and a more straightforward approach to parallel programming but, in general, the trade-off between programmability and performance has been favorable for the UPC code.

Acknowledgements

This work was funded by the Ministry of Science and Innovation of Spain (Project TIN2010-16735), the Spanish network CAPAP-H3 (Project TIN2010-12011-E), and by the Galician Government (Pre-doctoral Program "María Barbeito" and Program for the Consolidation of Competitive Research Groups, ref. 2010/6).

References

1. Beenakker, C.: Ewald Sum of the Rotne-Prager Tensor. Journal of Chemical Physics **85**, 1581–1582 (1986)
2. Długosz, M., Zieliński, P., Trylska, J.: Brownian Dynamics Simulations on CPU and GPU with BD_BOX. Journal of Computational Chemistry **32**(12), 2734–2744 (2011)
3. Ermak, D.L., McCammon, J.A.: Brownian Dynamics with Hydrodynamic Interactions. Journal of Chemical Physics **69**(4), 1352–1360 (1978)
4. Fixman, M.: Construction of Langevin Forces in the Simulation of Hydrodynamic Interaction. Macromolecules **19**(4), 1204–1207 (1986)
5. García de la Torre, J., Hernández Cifre, J.G., Ortega, A., Schmidt, R.R., Fernandes, M.X., Pérez Sánchez, H.E., Pamies, R.: SIMUFLEX: Algorithms and Tools for Simulation of the Conformation and Dynamics of Flexible Molecules and Nanoparticles in Dilute Solution. Journal of Chemical Theory and Computation **5**(10), 2606–2618 (2009)

6. Hipp, M., Rosenstiel, W.: Parallel Hybrid Particle Simulations Using MPI and OpenMP. In: 10th International Euro-Par Conference (Euro-Par'04), Pisa (Italy), pp. 189–197 (2004)
7. Huber, G.A., McCammon, J.A.: Browndye: A Software Package for Brownian Dynamics. Computer Physics Communications **181**(11), 1896 – 1905 (2010)
8. Klenin, K., Merlitz, H., Langowski, J.: A Brownian Dynamics Program for the Simulation of Linear and Circular DNA and Other Wormlike Chain Polyelectrolytes. Biophysical Journal **74**(2 Pt 1), 780–788 (1998)
9. Kloeden, P.E., Platen, E.: Numerical Solution of Stochastic Differential Equations. Springer, Berlin (1992)
10. Li, B., Zhu, Y.L., Liu, H., Lu, Z.Y.: Brownian Dynamics Simulation Study on the Self-assembly of Incompatible Star-like Block Copolymers in Dilute Solution. Physical Chemistry Chemical Physics **14**, 4964–4970 (2012)
11. Mallón, D.A., Taboada, G.L., Teijeiro, C., Touriño, J., Fraguela, B.B., Gómez, A., Doallo, R., Mouriño, J.C.: Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. In: 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09), Espoo (Finland), pp. 174–184 (2009)
12. Markidis, S., Lapenta, G.: Development and Performance Analysis of a UPC Particle-in-Cell Code. In: 4th Conference on Partitioned Global Address Space Programming Models (PGAS'10), New York (NY, USA), pp. 10:1–10:9. ACM (2010)
13. Rotne, J., Prager, S.: Variational Treatment of Hydrodynamic Interaction in Polymers. Journal of Chemical Physics **50**(11), 4831–4837 (1969)
14. Tocino, A., Vigo-Aguiar, J.: New Itô–Taylor Expansions. Journal of Computational and Applied Mathematics **158**(1), 169–185 (2003)
15. Unified Parallel C at George Washington University: http://upc.gwu.edu. Accessed 31 October 2012
16. Wei, W., Al-Khayat, O., Cai, X.: An OpenMP-enabled Parallel Simulator for Particle Transport in Fluid Flows. In: 11th International Conference on Computational Science (ICCS'11), Singapore, pp. 1475–1484 (2011)
17. Zhao, W., Wang, Z.: ScaleUPC: A UPC Compiler for Multi-core Systems. In: 3rd Conference on Partitioned Global Address Space Programing Models (PGAS'09), Ashburn (VA, USA), pp. 11:1–11:8. ACM (2009)