ExaStamp: A Parallel Framework for Molecular Dynamics on Heterogeneous Clusters

Emmanuel Cieren[1], Laurent Colombet[1], Samuel Pitoiset[2], and Raymond Namyst[2]

[1] CEA, DAM, DIF, F-91297 Arpajon, France
[2] Université de Bordeaux, INRIA, 351 Cours de la Libération, 33405 Talence Cedex, France

Abstract. Recent evolution of supercomputer architectures toward massively multi-cores nodes equipped with many-core accelerators is leading to make MPI-only applications less effective. To fully tap into the potential of these architectures, hybrid approaches – mixing MPI, threads and CUDA or OpenCL – usually meet performance expectations, but at the price of huge development and optimization efforts.

In this paper, we present a programming framework specialized for molecular dynamics simulations. This framework allows end-users to develop their computation kernels in the form of sequential-looking functions and generates multi-level parallelism combining vectorized and SIMD kernels, multi-threading and communications. We report on preliminary performance results obtained on different architectures with widely used force computation kernels.

Keywords: Molecular dynamics, MPI, threads, TBB, vectorization, OpenCL, object-oriented design, Lennard-Jones, EAM.

1 Introduction

Molecular dynamics (MD) is a method used to compute the dynamical properties of a particles system, widely spread in fields such as Materials Science, Chemistry and Biology. With its scalable structure, MD took a substantial step with the ever increasing computer capabilities: after starting at a few hundreds particles [1], MD simulations have successfully coped with million particles systems in the 90s [11], before reaching one billion particles in 2005 [9].

Parallelism in most MD codes is limited to classical domain-decomposition techniques, and the use of accelerators is still rare. In the same time, future processor architectures are expected to feature a large number of cores with a fair decrease of the available memory per core, and the use of a co-processor has become quite common. The Intel® Xeon Phi™ architecture illustrates this trend well.

Stamp is a classical molecular dynamics production code which has been developed at CEA for twenty years [18]. Its flat MPI architecture and the absence of vectorization will obviously not fit requirements of next generation processors. To the best of our knowledge, no existing MD program is able to exploit clusters of such hybrid nodes, potentially equipped with different accelerators, in a uniform way. The development of a new object-oriented framework ExaStamp, capable of fulfill these new needs, began in 2012.

Optimized for large scale simulations of solid-state materials and shock physics, this framework supports several levels of parallelization. Besides the classical hybrid programming model, we developed a tool which enable generation of efficient vectorized code and *OpenCL* kernels for modern CPUs, GPUs and Intel® Xeon Phi™ accelerators. The complexity of implementing different parallelisms has been hidden from the non-expert developer through its object-oriented design. For main algorithms, our framework contains parallelism in specific modules. In the case of compute-intensive parts, specific vectorized instructions can be instantiated from the same sequential-looking code. Furthermore, data structures and their associated algorithms were carefully designed so as to keep the memory footprint as low as possible, meeting the requirements of future many-core architectures.

This paper discusses the design, implementation and performance of ExaStamp framework. It is organized as follows: in Sect. 2 we introduce Molecular Dynamics simulations and present the classical parallelization approaches. The design and implementation of our approach are are presented in Sect. 3. Performance results on different computers architectures are detailed and analyzed in Sect. 4. Finally, some conclusions and perspectives are discussed.

2 Molecular Dynamics

The main principle of MD consists in numerically integrating Newton's equation of motion $\boldsymbol{f} = m\,\boldsymbol{a}$, where the force on a particle depends on the interactions with all others [2]. Among the multiple ways to solve this equation, the *Leapfrog* integrator and the *Verlet* integrator, which are equivalent, are the most used as they offer greater stability, as well as other properties, for a low computational cost [14].

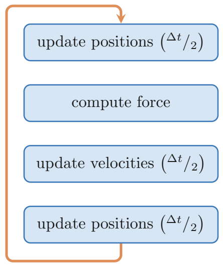

update positions $(^{\Delta t}/_2)$
compute force
update velocities $(^{\Delta t}/_2)$
update positions $(^{\Delta t}/_2)$

Fig. 1. Overview of a time-step in a MD simulation using the Leapfrog integrator

In most MD simulations, particles are treated as points and the interacting force between particles is approximated as a gradient of a potential that depends on the distance between those particles. The force computation is obviously the most challenging part: it contains all the physics of the simulation and can take up to 95% of the total time. When this potential comes from quantum mechanics principles, we talk about *ab initio* molecular dynamics; in the other case, the term *classical* molecular dynamics is used. Potentials from classical MD are empirical or semi-empirical; they are computed from an analytical formula, or they can be interpolated from tabulated values [22]. In this paper, we will focus on short-range interactions: it means that beyond a given distance r_c called the *cutoff* distance, interactions will be neglected. This approximation is completely justified for solid materials, since distant atoms are "screened" by nearer atoms. In case of systems with electrostatic or gravitational effects, long-range interactions cannot be omitted and special algorithms have been designed [10].

Although it was first designed to study gases, the *Lennard-Jones* potential (LJ) [13] has been used in a large part of material science, and became a standard benchmark for MD codes. The LJ potential is a pair potential, which means that it describes the interaction between a pair of particles (within the cutoff distance). For this potential, the expression of the energy on a particle i is given by

$$E_i = \frac{1}{2}\sum_j V(r_{ij})\,,\ \text{with}\ V(r_{ij}) \overset{r_{ij}\le r_\mathrm{c}}{=} 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right], \qquad (1)$$

where ε and σ are parameters which denotes respectively the well depth and the bond length.

Yet, pair potentials remain limited when it comes to bonded interactions: as an example, Stillinger and Weber developed a three-body potential for Silicon crystals [19]. For the study of metals and their alloys, effects from the electron charge density have a significant impact: the *Embedded Atom Method* (EAM) provides an accurate model and an acceptable computational cost [6,8,7].

$$E_i = \frac{1}{2}\sum_j \phi(r_{ij}) + F\left(\sum_j \rho_j(r_{ij})\right), \qquad (2)$$

where ϕ is a simple pair potential, ρ_j the contribution of the electron density near atom j, and F an embedding function representing the amount of energy required to place atom i in the electron cloud. Both ϕ and ρ are canceled beyond the cutoff distance. Common EAM potentials are for instance the Johnson potential, the *Sutton-Chen* (SC) and the Tersoff potentials [12,20,21].

3 A Framework for Molecular Dynamics Simulations

ExaStamp has been designed to replace the production code Stamp on the next generation of supercomputers. Targeting solid-state material and shock physics studies, it should be able to perform very large scale simulation of complex systems (a billion particles with many-body potentials) on a various range of architectures. As a future production code, all the programming refinement should be hidden from standard developers. To this end, we chose the C++ language and widely used C++11 standard features. With upcoming parallel architectures in mind, we also focused on minimizing memory footprint of our data structures, so as to handle large sets of particles.

There are three basic ways to parallelize work in a MD simulation: parallelization over particles, parallelization over pairs of particles and domain-decomposition. As explained in [16], the first two proved inefficient, as they require to many communications over the interconnection network, leaving the third one as the only possibility despite potential load-balancing issues.

The latter method is typically used in MPI implementations: the global domain is split and each process is assigned to a sub-domain. To compute interactions on the edges, each sub-domain will be enclosed in a ghost layer, which consists in a copy of the boundaries with its neighboring sub-domains. In practice, the length of this ghost layer is generally the cutoff radius. The outline in Fig. 1 is hardly modified: everything is performed in parallel, one extra step is used to send and receive particles moving between sub-domains, and another one to update the ghost layer.

3.1 Overall Parallelization Strategy

In our approach, the global domain is *overdecomposed* with respect to the underlying cluster nodes (as illustrated in Fig. 2). Several domains can thus be assigned to a single node, each being treated either by regular CPU cores or by accelerators. We now present the main concepts and algorithms used in our framework.

Node and Communication Manager. A `Node` is the top structure of the code. We decided to use this terminology as we intend to use one `Node` structure per machine node in production mode, so that we can take advantage of shared-memory systems. Thus it contains the integration scheme, a list of one or several domains, and a communication manager. The `Node` is also responsible for IOs.

The Communication manager structure is an object-oriented framework for communications. It allows a developer to create its own custom types and provides wrappers to use these types in communication.

Integration Scheme. The family of integration schemes depicted in Fig. 1 reveals that they are basically made of the same elementary functions: updating a quantity (particles positions or velocities) with an explicit (first or second order) Euler scheme, or the force computation. Therefore we can define a `NumericalScheme` as an object with a function `oneStep()`, which contains a sequence of predefined elementary functions. Implementing a new scheme does not require the knowledge of lower classes implementation, as long as the requested elementary functions are implemented.

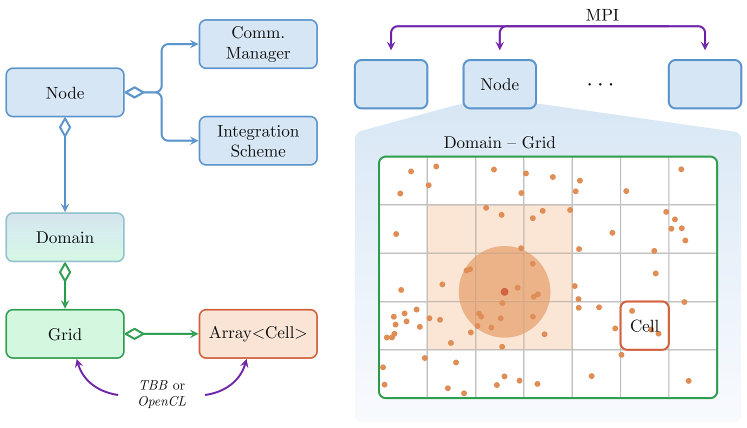


Fig. 2. Overview of ExaStamp architecture: pseudo-UML diagram with main classes of the code (left) with “physical” representation (right). The dark orange circle corresponds to the area of influence of a particle, whereas the light orange zone is the set of cells where neighbors will be looked for

Domain. Domain concept gathers an interface and its possible implementations. Domain interface contains basic accesses and elementary functions required by all `NumericalScheme` objects. A `Domain` proceeds to a reorganization and code factorization of these requirements for their implementation in lower-level classes.

Let us consider the force computation example. The code provides the possibility to overlap communication with computation: it means that it is possible to start updating ghost layers and compute forces inside the domain while communications are processing. Once the ghost layer update is over, we can start the force computation on the domain's edges. A `NumericalScheme` object does not need to know whether communication overlap has been enabled, it just asks for the forces computation. In the `Domain` class, two different functions handled by a strategy pattern are available.

Grid and Cells. When it comes to the force computation, each particle should get a list of its neighbors. Let us partition the domain with a virtual cubic mesh with a size slightly greater than the cutoff radius. Given a particle, we only have to look for its neighbors in the cell where it lives and its neighboring cells, which makes a total of 27 cells (in a three-dimensional space) to explore. This how the linked-list cell method [2] starts, reducing a naive pair search in $O(N^2)$ into a $O(N)$ algorithm.

Though we will not use this method (linked-cell list are not well suited neither for parallelism nor for vectorization), we will fully benefit from the cell partitioning. The task of `Grid` object is to implement all services required by the `Domain` class. In order to keep a high level of modularity without paying the cost of virtualization, we used a *curiously recurring template pattern* [5] in its implementation. Apart from this, we could almost reduce the `Grid` object into an array of `Cell` objects, which is where particles live.

We decided to focus the thread parallelism on this `Cell` array: it is roughly the same idea than a parallelization over particles, with a bigger grain-size. To maximize threads efficiency, we chose to store particles as structure of arrays (SOA) at the `Cell` level, which will become an array of structures of arrays (AOSOA) at the `Grid` level. Indeed, this structure enable vectorization within cells and is especially efficient when it comes to concurrent accesses: two threads working on two different cells can add and remove particles from those cells (which potentially means data reallocation). On the contrary, parallelization over one big array of particles would have required critical regions, throwing away any goal of performance on a many core system.

3.2 Code Specialization

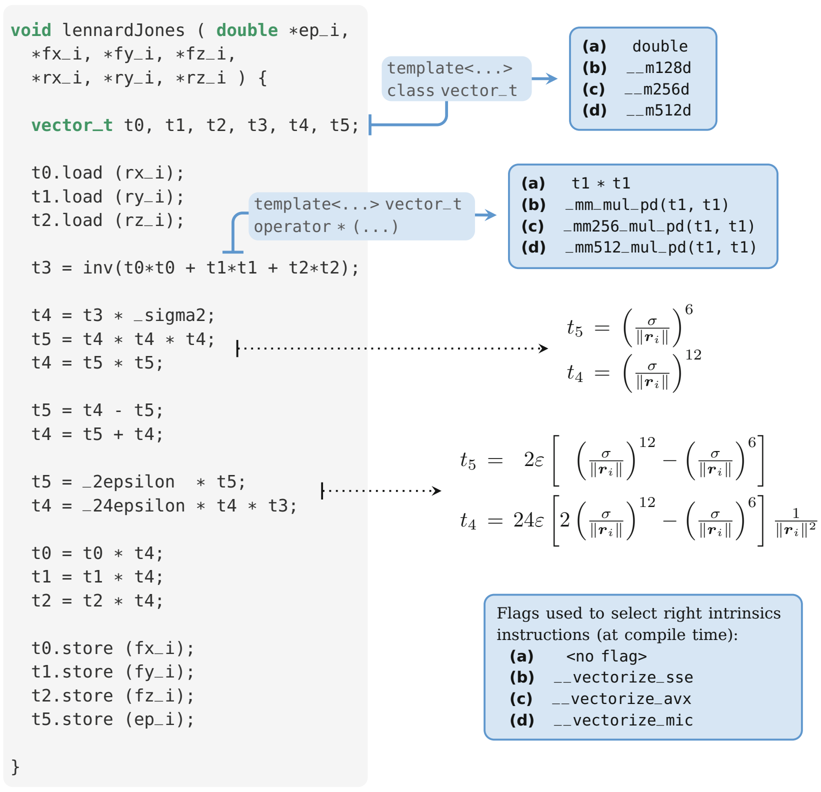
Performing high performance molecular dynamics over hybrid machines requires to use highly optimized computation kernels combining threads/tasks and vectorization over CPU cores or Intel® Xeon Phi™accelerators, and highly parallel SIMD code for GPU accelerators. In our Framework, domains assigned to regular CPU cores are parallelized using Intel®'s *Threading Building Blocks* (TBB) [4], whereas domains assigned to GPU or Intel® Xeon Phi™accelerators rely on a series of OpenCL kernels which parallelize each step of an iteration loop.

Despite progress made by compilers regarding auto-vectorization, writing code to maximize the number of vectorization opportunities detected by the compiler remains a delicate process. Writing efficient OpenCL code is also a delicate task, and actually requires to perform target-specific (and even platform-specific) optimizations. Intel, AMD and NVIDIA programming guides, for instance, each suggest different optimizations which can actually lower performance on other platforms. For all these reasons, implementing a new particle interaction potential would normally require to develop and optimize multiple versions of the force computation kernel (Fig. 1), in multiple languages.

To solve this problem, our framework allows force computation steps to be written as a set of C++ sequential-looking functions, as illustrated on Fig. 3 for the LJ potential.

When instantiated on multi-core architectures, this code is transformed using C++ template classes to generate intrinsic vector functions instead of scalar operations, to guarantee that the force computation kernel is fully vectorized. A unique sequential-looking code is used, whatever the type of vectorization is performed (no vectorization, SSE, AVX, or IMCI[1]). The obtained vectorized kernel is used inside Cells and is called from within a sequential loop iterating over particles. At the upper level, each iteration step is parallelized using a TBB *parallel for* loop iterating over Cells (as described in Section 3.1).

[1] Intel® Initial Many Core Instructions, a set of vector instructions for the KNC.



```
void lennardJones ( double *ep_i,
  *fx_i, *fy_i, *fz_i,
  *rx_i, *ry_i, *rz_i ) {

  vector_t t0, t1, t2, t3, t4, t5;

  t0.load (rx_i);
  t1.load (ry_i);
  t2.load (rz_i);

  t3 = inv(t0*t0 + t1*t1 + t2*t2);

  t4 = t3 * _sigma2;
  t5 = t4 * t4 * t4;
  t4 = t5 * t5;

  t5 = t4 - t5;
  t4 = t5 + t4;

  t5 = _2epsilon  * t5;
  t4 = _24epsilon * t4 * t3;

  t0 = t0 * t4;
  t1 = t1 * t4;
  t2 = t2 * t4;

  t0.store (fx_i);
  t1.store (fy_i);
  t2.store (fz_i);
  t5.store (ep_i);

}
```

```
template<...>
class vector_t
```

```
(a)   double
(b)   __m128d
(c)   __m256d
(d)   __m512d
```

```
template<...> vector_t
operator * (...)
```

```
(a)   t1 * t1
(b)   _mm_mul_pd(t1, t1)
(c)   _mm256_mul_pd(t1, t1)
(d)   _mm512_mul_pd(t1, t1)
```

$$t_5 = \left(\frac{\sigma}{\|\boldsymbol{r}_i\|}\right)^6$$
$$t_4 = \left(\frac{\sigma}{\|\boldsymbol{r}_i\|}\right)^{12}$$

$$t_5 = 2\varepsilon\left[\left(\frac{\sigma}{\|\boldsymbol{r}_i\|}\right)^{12} - \left(\frac{\sigma}{\|\boldsymbol{r}_i\|}\right)^6\right]$$
$$t_4 = 24\varepsilon\left[2\left(\frac{\sigma}{\|\boldsymbol{r}_i\|}\right)^{12} - \left(\frac{\sigma}{\|\boldsymbol{r}_i\|}\right)^6\right]\frac{1}{\|\boldsymbol{r}_i\|^2}$$

Flags used to select right intrinsics instructions (at compile time):

```
(a)    <no flag>
(b)   __vectorize_sse
(c)   __vectorize_avx
(d)   __vectorize_mic
```

Fig. 3. Implementation of force and energy computation function using a LJ potential. If V denotes the potential as described in (1), we have to compute $e_i = \frac{1}{2}V(\|\boldsymbol{r}_i\|)$ and $\boldsymbol{f}_i = -\nabla_{\|\boldsymbol{r}_i\|}V(\|\boldsymbol{r}_i\|)$. Written in a C-like way (except for the function signature which contains templates and operator), it hides intrinsics functions enabled at compile-time with predefined flags.

When instantiated on accelerators, the sequential version of the code (see variant *a* on Fig. 3) is called from within an OpenCL force computation kernel. This kernel is executed by as many OpenCL *workitems* as the number of particles in the domain. The generic part of the kernel is optimized either for GPUs (coalesced memory accesses, bank conflicts avoidance, weak code divergence) or for Intel® Xeon Phi™accelerators (vectorization, cache reuse), but all these optimizations are hidden to the end-user. In the next Section, we present the performance achieved by our framework on various hardware platforms.

4 Performance Evaluation

All tests in this Section have been performed on CCRT's clusters[2] (see Table 1 for CPU specifications – the GPU used for OpenCL test is a NVIDIA Tesla K20c).

[2] Centre de Calcul Recherche et Technologie – `http://www-hpc.cea.fr/en/complexe/ccrt.html`

Table 1. Specifications of CPU used for our different tests. Cache size displayed are L3 sizes, except for the KNC which is L2. Airain's Ivybridge and Standard partitions are respectively made of 360 and 594 nodes connected with an Infiniband QDR network.

	Airain		**Cirrus**
	Ivybridge	Standard	KNC
Model	Intel® Xeon® CPU E5-2680 v2	Intel® Xeon® CPU E5-2680	Intel® Xeon Phi™ Coprocessor 5120D
Max Freq. (GHz)	2.8	2.7	1.05
Number of cores	2×10	2×8	60
Cache Size (MB)	25.60	20.48	30.00
Vectorization	AVX	AVX	IMCI

Code was compiled using Intel® compiler (version 14.0.2) with O3 optimization and vectorization enabled. Simulations involve a FCC lattice (a=0.354 nm) of copper at 600 K, using either a LJ (ε=0.583 eV, σ=0.227 nm and r_c=0.227 nm) or an analytic Sutton-Chen potential (c=33.2, $\varepsilon=2.25 \cdot 10^{-2}$ eV, a_0=0.327 nm, n=9.05, m=5.01 and r_c=0.729 nm).

4.1 Vectorization

On a Single CPU Core. To compare compilers auto-vectorization capabilities against hand-vectorized code, we use the sample code presented in Fig. 3 and generate both a naive version (variant a) and a SIMD version (variant b, c or d). Results in Table 2 clearly exhibit that hand-vectorization is mandatory to get high performance on non-trivial computation kernels.

Fig. 4 presents vectorization performance over a full simulation, for two potentials: a light one (LJ) and an expensive one (SC, with analytical functions). As expected, the use of vector units is still quite efficient, especially for the SC potential (which is about 40% faster). Its vectorization has been made possible

Table 2. Performances of our "SIMD" wrapper against a naive version for a LJ potential. Here we compare execution times in seconds (average on a million runs with arrays of size 256) of both versions for different vectorization modes. Tests were performed on an Ivybridge (first three lines) and a Intel® Xeon Phi™ (last two lines).

Mode	**Flags**			**Naive**	**Simd**	**Speedup**
Default	`-O3`			2.42	2.26	1.07
SSE	`-O3`	`-msse4.1`	`-D__vectorize-sse`	2.41	1.06	2.27
AVX	`-O3`	`-mavx`	`-D__vectorize-avx`	2.48	0.73	3.39
Default	`-O3`			46.80	36.24	1.29
IMCI	`-O3`	`-mmic`	`-D__vectorize-mic`	46.79	5.10	9.17

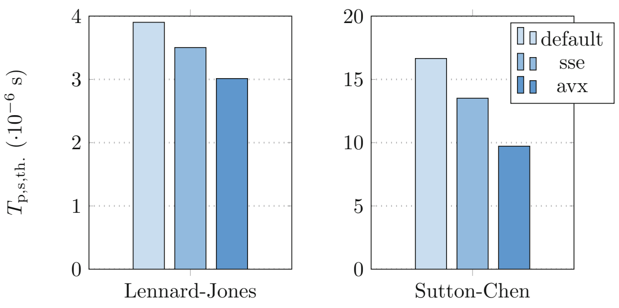


Fig. 4. Effect of vectorization for different potentials on Ivybridge (Airain). Simulations performed on 128 time-steps with one million atoms. $T_{\text{p,s,th.}}$ represents the time per particle per iteration per thread.

thanks to Intel® Short Vector Math Library (SVML), which provides intrinsic instructions for advanced math functions. Issues between Intel® compiler and C++11 standard prevented us from performing full native code simulations on a Intel® Xeon Phi™.

4.2 Multithreading

In Fig. 3, we compared memory usage for different number of threads and MPI processes. Memory usage was measured with the `getrusage()` function given by the standard C library, and sum across processes when needed. If it remains constant for simulations using only TBB, we observe that those which use only MPI ones need up to 25% more memory. Differences are even more important on a larger runs: for 4.3 billion particles on 16,384 cores, simulations with respectively 1, 2 and 4 threads per MPI process need 11.5, 9.5, and 8.3 Terabytes of memory, which make the full MPI about 40% more expensive.

Table 3. Comparison of maximum memory usage (in GB) between MPI and threads simulations on one Ivybridge node. Simulations performed on 64 steps with a SC potential.

Total Num. Cores	**1**	**10**		**20**	
MPI × Threads	1 × 1	10 × 1	1 × 10	20 × 1	1 × 20
$2 \cdot 10^6$ atoms	3.82	4.35	3.85	4.80	3.87
$5 \cdot 10^6$ atoms	9.46	10.12	9.51	10.82	9.59
$10 \cdot 10^6$ atoms	18.86	19.77	18.97	20.90	19.06

4.3 Scalability

Results from a weak scaling test up to 2,048 cores for different number of threads are plotted on Fig. 5. If the 16 threads case is obviously out of touch, it can be explained by NUMA accesses between sockets. From 1 to 8 threads the efficiency drop is very well contained, with all values between 90 and 95% for 2,048 cores. Runs with more than one thread are faster than the full-MPI one, although efficiency values are very close. It seems tricky to establish a clear hierarchy.

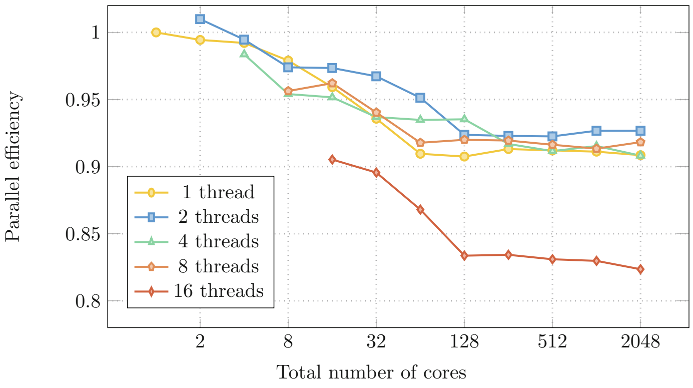


Fig. 5. Scaling tests of ExaStamp for different number of threads on Airain's Standard nodes. Simulations performed on 1,024 time-steps with $4.0 \cdot 10^5$ atoms per core using a SC potential.

4.4 Performances on Accelerators

Fig. 6 reports results obtained with the LJ potential on a Intel® Xeon Phi™ and a GPU. We observe that the GPU needs only five millions atoms to reach its peak performance, when the Intel® Xeon Phi™ requires around twenty. In single precision, the Intel® Xeon Phi™ gets slightly better performances (10% faster than the GPU), and this difference increases in double precision mode (+20%). Memory usage is perfectly linear with the number of atoms. As expected, the double precision mode requires twice the amount used for single precision one.

5 Related Work

Developed at Sandia National Laboratories, the LAMMPS [16] package has become a reference in MD. It can perform simulations up to a billion atoms on $64,000$ cores (using mainly MPI), covering physics from solid-state materials to soft matter. Yet, its multithreads implementation is still limited to some modules, with no better performance than full MPI [17, Sect. 5]. Gromacs [3] and NAMD [15] are more recent high-performance oriented codes targeting biomolecular systems, which is far from condensed matter physics. As a result, these programs require a completely different coding approach than ours.

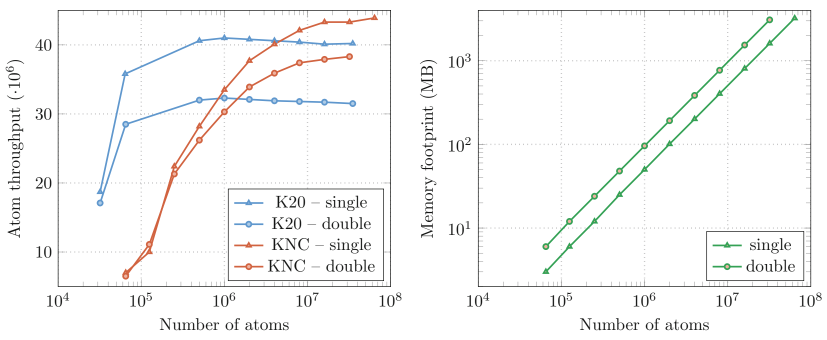


Fig. 6. Performance of OpenCL simulations on different accelerators, using LJ potential. For both single and double precision, we compare performance in term of atom throughput (number of atoms per second per iteration) and memory footprint.

6 Conclusion and Future Work

We presented ExaStamp, a classical molecular dynamics framework designed for production on new generation supercomputers. Its object oriented design allowed us to hide complexity introduced by multiple levels of parallelism. On that point, early returns by developers are very positive. Besides, performance results in terms of vectorization, scaling and memory usage are very promising.

We will soon be able to start testing on ExaStamp with native code on Intel® Xeon Phi™, which will undoubtedly be an important platform to achieve "real physics" simulations. On top of development of new potentials and numerical modules, we will also focus on the development of a dynamic load balancing capability on nodes level.

Acknowledgment. This work is integrated and supported by the PERFCLOUD project, a French FSN[3] cooperative project that associates academic and industrial partners to design and provide building blocks for new generations of HPC data-centers.

References

1. Alder, B.J., Wainwright, T.E.: Phase Transition for a Hard Sphere System. The Journal of Chemical Physics 27(5), 1208–1209 (1957)
2. Allen, M., Tildesley, D.: Computer Simulation of Liquids. Clarendon Press (1987)
3. Berendsen, H., van der Spoel, D., van Drunen, R.: GROMACS: A Message-passing Parallel Molecular Dynamics Implementation. Computer Physics Communications 91(1-3), 43–56 (1995)

[3] Fond pour la Société Numérique.

4. Contreras, G., Martonosi, M.: Characterizing and Improving the Performance of Intel Threading Building Blocks. In: IEEE International Symposium on Workload Characterization, IISWC 2008, pp. 57–66 (2008)
5. Coplien, J.O.: Curiously Recurring Template Patterns. C++ Rep. 7(2), 24–27 (1995)
6. Daw, M.S., Baskes, M.I.: Embedded-atom Method: Derivation and Application to Impurities, Surfaces, and other Defects in Metals. Phys. Rev. B 29, 6443–6453 (1984)
7. Daw, M.S., Foiles, S.M., Baskes, M.I.: The embedded-atom Method: a Review of Theory and Applications. Materials Science Reports 9(7-8), 251–310 (1993)
8. Foiles, S.M., Baskes, M.I., Daw, M.S.: Embedded-atom Method Functions for the FCC Metals Cu, Ag, Au, Ni, Pd, Pt, and their Alloys. Phys. Rev. B 33, 7983–7991 (1986)
9. Germann, T.C., Kadau, K., Swaminarayan, S.: 369 Tflop/s Molecular Dynamics Simulations on the Petaflop Hybrid Supercomputer 'Roadrunner'. Concurrency and Computation: Practice and Experience 21(17), 2143–2159 (2009)
10. Greengard, L., Rokhlin, V.: A Fast Algorithm for Particle Simulations. Journal of Computational Physics 73(2), 325–348 (1987)
11. Hoover, W.G., De Groot, A.J., Hoover, C.G., Stowers, I.F., Kawai, T., Holian, B.L., Boku, T., Ihara, S., Belak, J.: Large-scale Elastic-plastic Indentation Simulations via Nonequilibrium Molecular Dynamics. Phys. Rev. A 42(10), 5844–5853 (1990)
12. Johnson, R.A.: Alloy Models with the Embedded-atom Method. Phys. Rev. B 39, 12554–12559 (1989)
13. Jones, J.E.: On the Determination of Molecular Fields. II. From the Equation of State of a Gas. Proceedings of the Royal Society of London. Series A 106(738), 463–477 (1924)
14. Leimkuhler, B.J., Reich, S., Skeel, R.D.: Integration Methods for Molecular Dynamics. In: Mesirov, J.P., Schulten, K., Sumners, D.W. (eds.) Mathematical Approaches to Biomolecular Structure and Dynamics. The IMA Volumes in Mathematics and its Applications, vol. 82, pp. 161–185. Springer, New York (1996)
15. Nelson, M.T., Humphrey, W., Gursoy, A., Dalke, A., Kalé, L.V., Skeel, R.D., Schulten, K.: NAMD: a Parallel, Object-oriented Molecular Dynamics Program. International Journal of High Performance Computing Applications 10(4), 251–268 (1996)
16. Plimpton, S.: Fast Parallel Algorithms for Short-range Molecular Dynamics. Journal of Computational Physics 117(1), 1–19 (1995)
17. Sandia National Laboratories: LAMMPS User Manual (2014), `http://lammps.sandia.gov/doc/Manual.html`
18. Soulard, L.: Molecular Dynamics Study of the Micro-spallation. The European Physical Journal D 50(3) (2008)
19. Stillinger, F.H., Weber, T.A.: Computer Simulation of Local Order in Condensed Phases of Silicon. Phys. Rev. B 31, 5262–5271 (1985)
20. Sutton, A.P., Chen, J.: Long-range Finnis-Sinclair Potentials. Philosophical Magazine Letters 61(3), 139–146 (1990)
21. Tersoff, J.: New Empirical Approach for the Structure and Energy of Covalent Systems. Phys. Rev. B 37, 6991–7000 (1988)
22. Wolff, D., Rudd, W.G.: Tabulated Potentials in Molecular Dynamics Simulations. Computer Physics Communications 120(1), 20–32 (1999)