ExaNBody : a HPC framework for N-Body applications

Thierry Carrard, Raphaël Prat, Guillaume Latu, Killian Babilotte, Paul Lafourcade, Lhassan Amarsid, Laurent Soulard

ExaNBody : a HPC framework for N-Body applications

Thierry Carrard[1,2], Raphaël Prat[3], Guillaume Latu[3], Killian Babilotte[1,2], Paul Lafourcade[1,2], Lhassan Amarsid[3], and Laurent Soulard[1,2]

[1] CEA, DAM, DIF, F-91297 Arpajon, France
[2] Université Paris-Saclay, LMCE, 91680 Bruyères-le-Châtel, France
[3] CEA, DES, IRESNE, DEC, Cadarache F-13108 Saint-Paul-Lez-Durance, France

Abstract. Increasing heterogeneity among HPC platforms requires applications to be frequently ported and tuned, adding burden to developers. Fast evolution of hardware mandates adaptation of algorithms and data structures to get higher performance, while application complexity constantly grows accordingly. Ensuring portability while preserving high performance at large scale along with minimal changes to an already existing application is an actual challenge. Separation of concerns to decouple performance from semantics in simulation codes are typically required. We describe a specialized programming framework for N-Body simulations that provides such separation. It allows one to develop computation kernels in the form of sequential-looking functions, while self-generating multi-level parallelism. exaNBody possesses both an application layer with its own input data format, a way to define specific computation kernels and a separate runtime system that can address both CPUs and GPUs. The framework enables performance portability for N-Body simulations, bringing both flexibility and a set of handy tools. Performance results and speedups up to 32k cores with two distinct applications based on exaNBody are discussed.

Keywords: N-body simulation · HPC· MPI· OpenMP · Framework

1 Introduction

Collisional N-body simulations are extensively used in several scientific domains. One challenge is the realistic modeling of atomistic systems or particle-based and granular materials containing multi-million of elements. Discrete element method (DEM), classical molecular dynamics (MD), and smoothed-particle hydrodynamics (SPH) methods fall into this category of N-body approaches. N-body modeling becomes challenging as the computational domain size and complexity rise. An always renewed challenge for tools in this field is to take advantage from the increasing power of distributed memory machines in order to mimic real systems. As supercomputers complexity tends to increase, this requires to revisit algorithms in order to use computing units at their full potential. The exaNBody framework is modular, customizable, and allows to build a wide variety of different problems.

In this paper, we focus on the scalability of a set of numerical methods for performing N-body simulations embedded into a comprehensive framework offering genericity and portable performance.

exaNBody provides features for numerically integrating Newton's equation of motion for each particle, resulting from short/long range interactions or contact between objects (in the case of Dem simulations). Interactions computation methods rely on neighbor search algorithms and numerical time integration schemes.

To fasten developments, we propose a Hpc framework to build N-body applications. The main contributions of this paper are the following. First, we introduce the framework and its features to address N-Body problems in a flexible way. Then, the framework architecture, based on a Mpi+X parallelism is explained. Finally, exaNBody is evaluated throughout target applications on thousands of Cpu cores, as well as on a set of Gpus.

2 Background : N-body simulations

2.1 N-body methods

N-body methods encompass a variety of techniques used to model the behavior and interactions of a set of particles over time. These methods consist in solving Newton's equation of motion $\mathbf{f} = m\mathbf{a}$ for each particle at each time step, where $\mathbf{f}$ corresponds to the sum of the forces applied to the particle, $\mathbf{a}$ its acceleration and m its mass. The forces are deduced from the interactions between particles according to their types, *i.e.* contact, short-range, or long-range interactions, and external forces applied to the sample (i.e. gravity). Velocities are then deduced from the accelerations and subsequently used to update the particle positions at the next time step. This process is repeated, typically with a fixed time step Δ_t, according to an integration scheme[4] until the desired duration is reached. The collection of particle configurations over time allows to study a wide range of phenomena, from granular media movements, with the Discrete Element Method (Dem), to material crystal plasticity at the atomic scale using Molecular Dynamics (Md), going up to the galaxy formation with the Smoothed-Particle Hydrodynamics (Sph).

2.2 N-body simulation codes

The development of a N-body code is led by the need to figure out the neighborhood of a given particle for every timestep in order to process interactions of different kinds.

Particle interactions can be categorized as short-range and long-range. Short-range interactions are considered negligible beyond a specified cut-off radius. To optimize calculations, neighboring particle detection algorithms are employed to eliminate unnecessary computations with distant particles. Each N-Body method employs a wide variety of short-range interactions that capture different particle physics. For example, visco-elastic contacts in Dem follow

[4] For example, Velocity Verlet integrator can be used [13].

Hooke's law or Hertz law to model contact elasticity between rigid particles, while pair potentials like Lennard-Jones or Morse are used for gas or liquid atom interactions in classical MD. Long-range interactions, on the other hand, can sometimes not be neglected and result in algorithmic complexity of $\mathcal{O}(N^2)$. Such interactions, like gravitation in astrophysics, or electrostatic forces in MD, are typically modeled using the Ewald summation method. Fortunately, calculation approaches such as the fast multipole method can achieve a complexity of $\mathcal{O}(N)$, thanks to an octree structure, and can be efficiently parallelized [1]. Although this paper primarily focuses on short-range interactions, both types of interaction can be dealt with in EXANBODY.

Neighbor lists are built, using different strategies, to shorten the process of finding out the neighbors of a particle within the simulation domain. It helps optimizing the default algorithm having a complexity of $\mathcal{O}(N^2)$ that tests every pair of particles (if N is the number of particles). The most common strategy to deal with any kind of simulation (static or dynamic, homogeneous and heterogeneous density) is a fusion between the linked-cell [7] method and the Verlet list method [24]. The combination of these methods has a complexity of $\mathcal{O}(N)$ and a refresh rate that depends on the displacement of the fastest particle. This algorithm is easily thread-parallelized. Others less-used neighbor search strategies have been developed to address specific simulations, such as for static simulation with particles respecting a regular layout [10].

Domain decomposition is usually employed in N-Body methods to address distributed memory parallelization[17], assigning one subdomain to each MPI process. This implies the addition of ghost areas (replicated particles) around subdomains to ensure each particle has access to its neighborhood. Over time, many algorithms have been designed to improve load-balancing such as: Recursive Coordinate Bisection (RCB), the Recursive Inertial Bisection (RIB), the Space Filling Curve (SFC), or graph method with PARMETIS. Note that the library Zoltan gathers the most popular methods. To ease neighbor list construction (i.e. employing the linked-cell method), the simulation domain is described as a cartesian grid of cells, each of which containing embedded particles. Each subdomain then consists in a grid of entire cells assigned to one MPI process. In contrast, concurent iteration over the cells of one subdomain's grid provides the basis for thread parallelization at the NUMA node level.

Commonalities are shared across N-body simulation codes, such as numerical schemes, neighbor particle detection, or short/long-range interactions. The computation time dedicated to interaction and force calculations can be significant (over 80% of the total time) depending on the studied phenomenon complexity. Additional factors, such as neighbor lists construction computational cost, can impact overall simulation time. For dynamic simulations involving rapidly moving particles and computationally inexpensive interactions, more than 50% of the total time may be spent on neighbor list construction. The computationally intensive sections of the code vary depending on methods and phenomena studied, requiring optimizations such as MPI parallelization, vectorization, multi-threading, or GPU usage.

A short review of N-body HPC codes shows that a significant amount of research has been devoted to adapting N-Body specific optimizations to supercomputer architectures evolutions. One of the most significant code in the scientific community is the state-of-the-art MD code LAMMPS [22]. LAMMPS has been continuously developed for nearly three decades and includes MPI+X parallelization using native languages such as OPENMP or CUDA. An interesting package of LAMMPS is the package LIGGGHTS [12] which reuses data structures of LAMMPS to perform DEM simulations. Others widely used MD codes are GROMACS and NAMD working with a hybrid MPI+X parallelization. Although more confidential, we introduce here EXASTAMP [8,18], a MD code that has demonstrated twice the performance of the LAMMPS code on micro-jetting case composed of billions of particles [19]. Several codes are devoted to DEM applications such as MERCURYDPM including a hybrid parallelization MPI+OPENMP and non open source software like EDEM and ROCKYDEM (including multi-GPU parallelization). Overall, the HPC community has put a lot of efforts in parallelizing those physics codes on thousands of cores, as for the SPH method on both CPU [16] and GPU [6], or the DEM [12].

3 Parallel programming models

With the emergence of heterogeneous HPC platforms and the intensive use of GPUs, the HPC community has developed portable solutions to help developers optimize their codes on a wide variety of supercomputers. These solutions can be classified into four catgories: (1) libraries proposing a set of parallel routines and equipped with several back-ends, (2) high-level directive-based instructions, (3) algorithms accessible through a programming language with parallel execution policies, and (4) Domain Specific Language (DSL).

Two commonly used parallel libraries are KOKKOS [23] and RAJA [4]. For instance, KOKKOS is one of the available parallel back-end in LAMMPS and achieves similar multi-threaded performances compared to the OPENMP back-end [11] while being portable on GPU. Note that KOKKOS does not manage MPI level parallelization. Similarly, RAJA currently proposes back-ends support for OPENMP, TBB, SIMD, CUDA, HIP, OPENMP target offloading and SYCL. KOKKOS and RAJA provide high-level abstractions for expressing the parallel constructs that are mapped onto a runtime to achieve portable performance.

Although these programming models propose a high-level portability, the performance penalty is low but not negligible. Indeed, Martineau et al. [14,15] have reported a penalty from 5 to 30% using KOKKOS and RAJA against OPENMP, CUDA or OpenCL version. Artigues et al. [2] have evaluated the performance portability for a Particle-In-Cell (PIC) code using KOKKOS and RAJA on V100 GPU. They concluded that KOKKOS and RAJA are at least twice longer than the CUDA version to carry out the calculations while the KOKKOS version was about 14% slower than the OPENMP version on CPU. A tuning step is sometimes expected to improve performance on GPUs. On the other hand, less-intrusive programming models than KOKKOS-like solutions exist such as the directive-based programming models (2) like OPENMP for thread parallelism on CPU,

or OpenMP and OpenACC on Gpu. Nevertheless, the use of programming directives often requires a non-trivial tuning process according to the computing platform considered.

stdpar is the C++ standard's parallel programming model (3) targeting both Cpu and Gpu. stdpar exposes parallel versions of the STL's main algorithms like *std::for_each* , expressing potential parallelism through an execution policy, such as *std::execution::par_unseq*. stdpar has been tested on Gpu and achieved similar performances compared to Kokkos or OpenMP for some mini-apps [3] and is less intrusive. Thrust and Boost propose similar approaches.

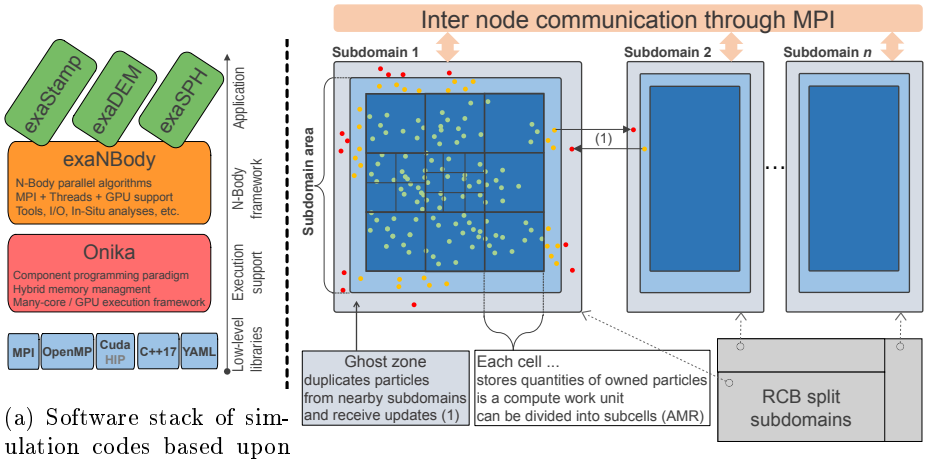
Finally, DSL based solutions (4) for N-body problems has been less investigated by the Hpc community. Beni et al. [5] propose the unique DSL (to our knowledge) to solve N-body problems with a slight average runtime overhead of 5%. Overall, DSL has the advantage to drastically reduce the number of lines while proposing a very high-level of abstraction. Nevertheless, this DSL does not include Mpi parallelization which is a major limiting factor. While identifying an efficient, flexible and portable way to model N-body problems on current supercomputers is still an open issue; our contribution aims at providing HPC optimized software shared by a wide variety of N-Body problems in a framework between a DSL and an ad-hoc N-Body code using native languages and tools.

4 Contribution

exaNBody offers a user-friendly and practical solution to harness the power of cutting-edge supercomputers. Its flexibility enables easy extension and specialization to meet specific hardware architectures and application requirements.

4.1 ExaNBody in a nutshell

exaNBody is a software platform developed at the french Alternative Energies and Atomic Energy Commission (CEA) for N-body problems involved in different fields of physics. Originally designed as part of exaStamp, a Md code for atomistic simulation, exaNBody has achieved nearly linear speedups on thousands of cores (100,000+) for simulations involving highly heterogeneous density scenarios, such as droplet splashing [21] or micro-jetting [20], with half a trillion atoms. Customized algorithms and data structures have been developed to achieve these results, whether for storing neighbor lists or processing communications. These advancements have made the code base increasingly generic and customizable to accommodate a growing number of physical models, while also progressively supporting Gpu equipped supercomputers. As a result, the authors decided to extract the N-Body core, exaNBody, as a standalone project, making it available for other N-Body simulation codes. exaNBody now evolves alongside exaStamp, focusing on flexibility, performance and portability, using industry standards as its software stack basis and providing application-level customizability (see Fig. 1a). exaNBody encompasses various aspects of N-Body simulation codes construction. Firstly, it provides a high level of flexibility through a component-based programming model. These components serve as



(a) Software stack of simulation codes based upon exaNBody.

(b) Overview of domain decomposition and inter process communications in exaNBody framework.

Fig. 1: Overview of the exaNBody software stack (a) and coarse grain structure of a parallel application based upon it (b)

building blocks and are assembled using YAML formatted files. Secondly, it offers portable performance by providing developers with a collection of algorithms and programming interfaces specifically designed for common N-Body compute kernels. Additionally, the programming tools aforementioned are natively compatible with different CPU and GPU architectures, thanks to the Onika execution support library. Before diving into these features, let us now describe how a simulation developer starts setting up its application.

4.2 Application level specialization

First of all, the internal units to be used are specified as well as the physical quantities to be stored as particle attributes. These quantities (or *fields*), are defined using a symbolic name associated with a type, e.g. *velocity* as a 3D vector. A *field set* is a collection of declared fields. One of the available *field sets* is selected and used at runtime, depending on simulation specific needs. As depicted in Fig. 1b, particles are dispatched in cells of a cartesian grid spanning the simulation domain. In short, the data structure containing all particles' data will be shaped as a cartesian grid of cells, each cell containing all *fields* for all particles it (geometrically) contains. More specifically, the reason why *fields* and *field sets* are defined at compile time is that particle data storage at the cell level is handled via a specific structure guaranteeing access performance and low memory footprint, detailed in section 4.4.

4.3 Flexible and user friendly construction of N-Body simulations

A crucial aspect for software sustainability is to maintain performance over time while managing software complexity and evolution. Complex and rapidly evolving scientific software often encounter common pitfalls, such as code duplication,

uncontrolled increase in software inter-dependencies, and obscure data/control flows. This observation has led us to develop our component-based model to avoid these pitfalls. In our model, individual software components are implemented using C++17 and are application structure oblivious, meaning they only know their input and output data flows. Application obliviousness is a crucial aspect of the present design, promoting reusability while preventing uncontrolled growth of internal software dependencies. Each component is developed as a class, inheriting from a base class *OperatorNode* and explicitly declares its input and output *slots* (data flow connection points). Once compiled, these components are hierarchically assembled at runtime using a Sequential Task Flow (STF) [1], with a YAML syntax, as shown in Fig. 2.

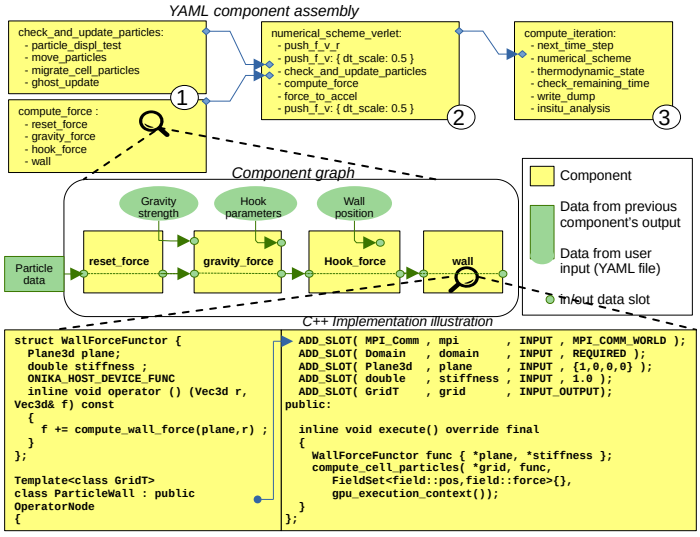


Fig. 2: Illustrative sample of components assembly using YAML description. 1) C++ developed components are assembled and connected in the manner of a STF, creating a *batch* component. 2) and 3) illustrate batch components aggregation to higher and higher level components, up to full simulation task flow.

A set of base components are already available to developers, embedded within EXANBODY, such as: common computations, checkpoint/restart, visualization and In-Situ analytics, allowing developers to focus on their application specific components. We also observed that this component based approach not only prevents some development pitfalls, but enables various simulation code structures. YAML formatted component configuration makes it simple for a user to amend or fine tune the simulation process. For instance it can be used to change the numerical scheme or even to insert In-Situ analysis components (such as proposed in [9]) at specific stages of the simulation process, leveraging In-Situ processing to limit disk I/O. Finally, this component based splitting of

the code gives exaNBody the opportunity to provide integrated profiling features that automatically give meaningful performance metrics for each part of the simulation. It allows the user to access computation time spent on Cpu and Gpu, as well as imbalance indicator. It can also interoperate with *nSight System* from NVIDIA and summarize memory footprint with detailed consumption.

4.4 Performance and portability

The complex and ever-changing architectures of modern supercomputers make it difficult to maintain software performance. exaNBody aims at providing performance portability and sustainability on those supercomputers with robust domain decomposition, automated inter-process communications algorithms, adaptable particle data layout, and a set of hybrid (Cpu/Gpu) parallelization templates specialized for N-Body problems.

Spatial domain decomposition and inter-process communications are critical to ensure scalability at large scales. Indeed, the coarsest parallelization level can become the main bottleneck due to network latencies and load imbalance issues. To take advantage of this first level of parallelization, the simulation domain is divided into subdomains using an Rcb algorithm, as depicted in Fig. 1b, assigning one subdomain to each Mpi process. This is achieved thanks to three main components: cell cost estimator, Rcb domain decomposition, and particle migration. Particle migration can be used as-is by any N-Body application, thanks to the underlying generic particle data storage (see Section 4.4). It supports heavily multi-threaded, large scale, simulations while lowering peak memory usage. Additionally, the migration algorithm is also customizable to fit specific application needs, keeping unchanged the core implementation. For instance, Md simulations may transport per-cell data fields and Dem simulations may migrate friction information related to pair of particles. Finally, ghost particle updates are available to any N-Body application, via customizable components.

Particle data layout and auxiliary data structures are two essential features to maximize performance at the NUMA node level. In exaNBody, particle data are packed independently in each cell using a container specifically designed to match both Cpu's Simd and Gpu's thread blocks requirements concerning data alignment and vectorization friendly padding. This generic container, available in Onika toolbox, not only adapts to specific hardware characteristics at compile time, but ensures minimal memory footprint with as low as 16 bytes overhead per cell regardless of the number of data fields, allowing for very large and sparse simulation domains. N-Body simulations also heavily depend on particles' neighbors search algorithm and storage structure. The search usually leverages the grid of cell structure to speed up the process, and neighbors lists data structure holds information during several iterations, see Section 2.2. However, depending on the simulation, particles may move rapidly while their distribution may be heterogeneously dense. Those two factors respectively impact neighbor list update frequency and its memory footprint. On the one hand, exaNBody takes advantage of an Adaptive Mesh Refinement (Amr) grid [18] to accelerate (frequent) neighbor list reconstructions. On the other hand, a compressed neighbor

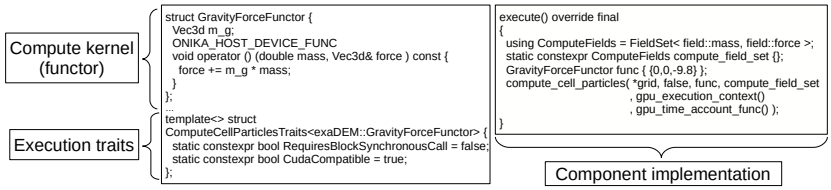


Fig. 3: Example of a particle centered computation executable on both CPU and GPU. Three ingredients: a user functor (the kernel), static execution properties (via traits specialization), a ready to use parallelization function template.

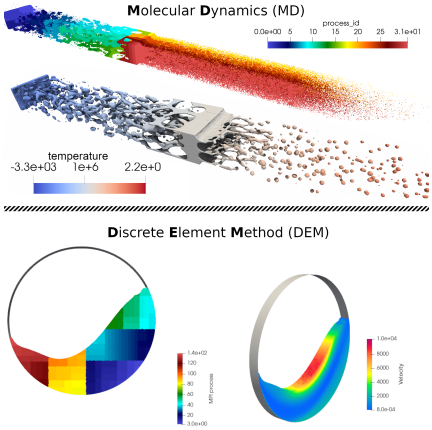
list data structure saves up to 80% of memory (compared to uncompressed lists) while still ensuring fast access from both the CPU and the GPU.

Intra-node parallelization API is available in EXANBODY to help developers express parallel computations within a MPI process. This API offers a set of parallelization templates associated with three types of computation kernels: local calculations on a particle, calculations coupled with reduction across all particles, and, most importantly, calculations involving each particle and its neighbors. When a developer injects a compute function into these templates, computation may be routed to CPU or GPU, as illustrated in Fig. 3. While thread parallelization on the CPU is powered by OPENMP, CUDA is employed to execute the computation kernel on the GPU, using the same provided function. The main difference between the two execution modes is that each cell is a unitary work unit for a single thread in OPENMP context but it is processed by a block of threads in CUDA. Those two parallelization levels (multi-core and GPU) are easily accessible to developers thanks to the execution support layer of Onika. Onika is the low-level software interface that powers EXANBODY building blocks. It is responsible for aforementioned data containers, memory management (unified with GPU), and it is the foundation for hybrid execution abstraction layer.

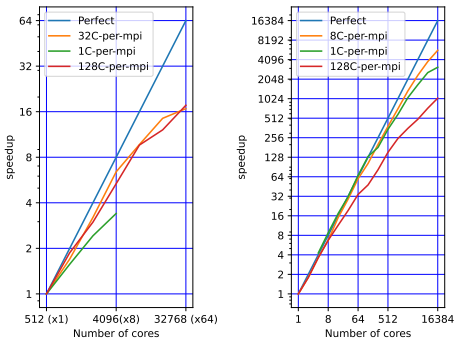
5 Numerical experiments: MD and DEM simulations

The present framework was evaluated with two applications : EXASTAMP, which employs MD, and EXADEM (coded with as few as 5500 lines) which relies on the DEM. Different OPENMP/MPI configurations (number of cores/threads per MPI process) have been tested to balance multi-level parallelism. Both simulations were instrumented during $1,000$ representative iterations. The performance of EXANBODY was evaluated using up to 256 cluster nodes, built on bi-socket 64-core AMD® EPYC Milan 7763 processors running at 2.45 GHz and equipped with 256 GB of RAM. We also ran CPU/GPU comparisons (512 CPU cores vs 16 NVidia A100 GPUs) to show GPU gains. Also, not included here, results show GPU gains of up to $\times 11$ on a100 versus one node depending on the force computation kernel.

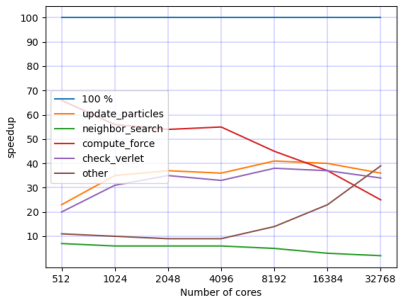
Molecular Dynamics performance is evaluated with the simulation of an impacted 640 million Tantalum atoms sample surrounded by air, leading to spal-



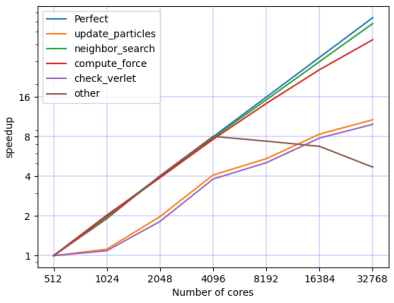
(a) Visualization of the MD simulation of metal spalling (upper) and the DEM simulation of spheres in a rotating drum.



(b) Speedup for different OpenMP/Mpi configurations. Left) MD simulation tested with 8, 32, and 128 threads per Mpi process. Right) DEM simulation with 1, 8, and 128 threads per Mpi process.



(c) Operator time ratios at different parallellization scales.



(d) Per operator speedup according to the total number of cores used.

Fig. 4: Results with two different applications : MD simulation of a 640 million atoms bulk and DEM simulation of 100 million spheres in a rotating drum.

lation due to shock-waves reflection on free surfaces (see figure 4a). This benchmark challenges MD application for two reasons: firstly, high velocity particles renders difficult to keep track of particle neighborhoods (Verlet list algorithm). Secondly, while Tantalum is a dense material, nucleation and growth of cavities creates large and increasing voided regions, leaving more and more cells empty (with no particle), making crucial subdomain partitioning for overall load balance. Moreover, the computation domain expands rapidly as the front of the bulk is propelled forward. The employed Modified Embedded Atom Model (MEAM) force model is another challenge: it involves second order neighbors, and must also be executed on ghost particles to get correct results. This intrinsically limits

Mpi scaling, because number of ghost particles drastically increase with smaller and smaller subdomains, and in turn prevents the 1 core per Mpi process to have decent performances. Despite this flaw, we observe speedup gains when scaling from 512 to 32,768 cores, although not ideal (*i.e.* linear). Left graph of figure 4b shows relative speedups for different Mpi/OpenMP configurations, ranging from 1 to 128 cores per Mpi process, with 32 cores per Mpi process being the best combination for this case. This configuration is detailed in figure 4d and figure 4c which highlight how different parts of the simulation scale and how their relative costs evolve.

Discrete Element Method performance is evaluated with a simulation of a rotating drum containing 100 million spherical particles, see Fig. 4a. This setup is a tough benchmark as particles are rapidly moving all around the heterogeneously dense domain, due to gravity. Additionally, the employed Hooke force model has a low arithmetic intensity, and exaDEM must handle pairwise friction information, that is updated by kernel and must migrate between Mpi processes when subdomains are redistributed. Those two characteristics highlight exaNBody framework overall overhead as well as its ability to fit different simulation methods. Different Mpi and OpenMP configurations were also tested, showing best performance with 8 threads OpenMP per Mpi process. This difference with Md demonstrates that Dem is less sensitive to subdomain decomposition and more sensitive to memory bandwidth, 8 being the number of cores sharing L3 cache. Note that compute operators (neighbors and compute), fall to respectively from 32.3% to 12.9% and from 43.5% to 13.1% for 1 and 16,384 cores whereas these operator speedups are almost perfect, respectively 14,281 and 18,943 (NUMA effect). The loss of performance is due to the expensive collective MPI functions (MPI_Allreduce), 4.74% for 128 cores, 18.8% for 2,048 cores, and 49.75% for 16,384 cores, that become predominant as the subdomains shrink. In conclusion, the application exaDEM based on exaNBody shows good parallel performance in strong scaling from 1 to 16,384 cores for a large scale simulation on a recent Hpc platform (Cpu cores are used).

6 Conclusion and Future Works

Simulation codes portability on modern Hpc platforms is increasingly complex and costly. As far as N-Body methods are concerned, the exaNBody framework presented here, driven by performance and portability, provides hybrid parallelization (Mpi + OpenMP + Gpu) application building blocks. Our solution is halfway between a general purpose library like Kokkos and a DSL. We have exhibited the concepts in exaNBody architecture that address specific needs of various types of N-body methods like Dem, Md, or Sph. exaNBody has been evaluated over largely unbalanced test cases in Md and Dem with up to 32,768 cores on strong scaling and shows continuous performance gains.

exaNBody should soon become open source and widely available. Before that, we want to compare current implementation of exaNBody core, using home grown Onika execution layer, with one based on Kokkos, Raja or stdpar in order to measure potential benefits of these tools regarding portability.

References

1. Agullo, E., et al.: Bridging the gap between OpenMP and task-based runtime systems for the fast multipole method. IEEE TPDS **28**(10), 2794–2807 (2017)
2. Artigues, V., et al.: Evaluation of performance portability frameworks for the implementation of a particle-in-cell code. CCPE **32**(11), e5640 (2020)
3. Asahi, Y., et al.: Performance portable Vlasov code with C++ parallel algorithm. In: IEEE/ACM Int. Workshop P3HPC. pp. 68–80. IEEE (2022)
4. Beckingsale, D.A., et al.: RAJA: Portable performance for large-scale scientific applications. In: IEEE/ACM Int. Workshop P3HPC. pp. 71–81. IEEE (2019)
5. Beni, L.A., et al.: Portal: A high-performance language and compiler for parallel n-body problems. In: IPDPS. pp. 984–995. IEEE (2019)
6. Cercos-Pita, J.L.: AQUAgpusph, a new free 3D SPH solver accelerated with OpenCL. CPC **192**, 295–312 (2015)
7. Ciccotti, G., Frenkel, D., Mc Donald, I.R.: Simulation of liquids and solids (1987)
8. Cieren, E., et al.: ExaStamp: a parallel framework for molecular dynamics on heterogeneous clusters. In: Euro-Par 2014 Workshops. pp. 121–132. Springer (2014)
9. Dirand, E., Colombet, L., Raffin, B.: Tins: A task-based dynamic helper core strategy for in situ analytics. In: SCFA 2018. pp. 159–178. Springer (2018)
10. Hu, C., et al.: Crystal MD: The massively parallel molecular dynamics software for metal with BCC structure. CPC **211**, 73–78 (2017)
11. Jeffers, J., et al.: Chapt. 20 - optimizing classical molecular dynamics in LAMMPS. In: Intel Xeon Phi Proc. High Perf. Prog. (2nd Edition), pp. 443–470 (2016)
12. Kloss, C., Goniva, C., Hager, A., Amberger, S., Pirker, S.: Models, algorithms and validation for opensource DEM and CFD–DEM. Progress in Computational Fluid Dynamics, an International Journal **12**(2-3), 140–152 (2012)
13. Leimkuhler, B.J., et al.: Integration methods for molecular dynamics. Mathematical Approaches to biomolecular structure and dynamics pp. 161–185 (1996)
14. Martineau, M., McIntosh-Smith, S., Boulton, M., Gaudin, W.: An evaluation of emerging many-core parallel programming models. In: Proceedings of the 7th Int. Workshop on PMAM. pp. 1–10 (2016)
15. Martineau, M., et al.: Assessing the performance portability of modern parallel programming models using TeaLeaf. CCPE **29**(15), e4117 (2017)
16. Oger, G., other: On distributed memory MPI-based parallelization of SPH codes in massive HPC context. CPC **200**, 1–14 (2016)
17. Plimpton, S.: Fast parallel algorithms for short-range molecular dynamics. Journal of computational physics **117**(1), 1–19 (1995)
18. Prat, R., et al.: Combining task-based parallelism & adaptive mesh refinement techniques in molecular dynamics simulations. In: Proc. ICPP. pp. 1–10 (2018)
19. Prat, R., et al.: AMR-based molecular dynamics for non-uniform, highly dynamic particle simulations. CPC **253**, 107177 (2020)
20. Soulard, L.: Micro-jetting: A semi-analytical model to calculate the velocity and density of the jet from a triangular groove. J. of App. Phys. **133**(8), 085901 (2023)
21. Soulard, L., Carrard, T., Durand, O.: Molecular dynamics study of the impact of a solid drop on a solid target. Journal of Applied Physics **131**(13), 135901 (2022)
22. Thompson, A.P., et al.: LAMMPS - a flexible simulation tool for particle-based materials modeling at atomic, meso, and continuum scales. CPC **271**, 108171 (2022)
23. Trott, C.R., et al.: Kokkos 3: Programming Model Extensions for the Exascale Era. IEEE TPDS **33**(4), 805–817 (2022)
24. Verlet, L.: Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. Physical review **159**(1), 98 (1967)