Raising the Performance of the Tinker-HP Molecular Modeling Package on Intel®'s HPC Architectures: a Living Review [Article v1.0]

Luc-Henri Jolly[1], Alejandro Duran[2], Louis Lagardère[1, 3], Jay W. Ponder[4], Pengyu Ren[5], Jean-Philip Piquemal[6, 7, 5]*

[1]Institut Parisien de Chimie Physique et Théorique, Sorbonne Université, FR 2622 CNRS, 75005, Paris France.; [2]Intel Corporation Iberia, Spain; [3]Institut des Sciences du Calcul et des Données, Sorbonne Université, 75005, Paris France. ; [4]Department of Chemistry, Washington University in Saint Louis, Saint Louis, Missouri 63130, United States; [5]Department of Biomedical Engineering, The University of Texas at Austin, Austin, Texas 78712, United States.; [6]Laboratoire de Chimie Théorique, Sorbonne Université, UMR 7616 CNRS, 75005, Paris France.; [7]Institut Universitaire de France, 75005, Paris, France

This LiveCoMS document is maintained online on GitHub at https://github.com/TinkerTools/Tinker-HP/tree/master/LiveCOMS; to provide feedback, suggestions, or help improve it, please visit the GitHub repository and participate via the issue tracker.

This version dated July 2, 2019

Abstract This living paper reviews the present High Performance Computing (HPC) capabilities of the Tinker-HP molecular modeling package. We focus here on the reference, double precision, massively parallel molecular dynamics engine present in Tinker-HP and dedicated to perform large scale simulations. We show how it can be adapted to recent Intel®Central Processing Unit (CPU) petascale architectures. First, we discuss the new set of Intel®Advanced Vector Extensions 512 (Intel AVX-512) instructions present in recent Intel processors (e.g., the Intel®Xeon®Scalable and Intel®Xeon Phi™2nd generation processors) allowing for larger vectorization enhancements. These instructions constitute the central source of potential computational gains when using the latest processors, justifying important vectorization efforts for developers. We then briefly review the organization of the Tinker-HP code and identify the computational hotspots which require Intel AVX-512 optimization and we propose a general and optimal strategy to vectorize those particular parts of the code. We intended to present our optimization strategy in a pedagogical way so it could benefit to other researchers and students interested in gaining performances in their own software. Finally we present the performance enhancements obtained compared to the unoptimized code both sequentially and at the scaling limit in parallel for classical non-polarizable (CHARMM) and polarizable force fields (AMOEBA). This paper never ceases to be updated as we accumulate new data on the associated Github repository between new versions of this living paper.

***For correspondence:**
jean-philip.piquemal@sorbonne-universite.fr (JPP)

1 Introduction

Tinker-HP is a massively MPI parallel package dedicated to classical molecular dynamics (MD) and to multiscale simulations, especially using advanced polarizable force fields (PFF) encompassing distributed multipoles electrostatics[1]. It is an evolution of the popular Tinker package code [2] which conserves its simplicity of use and its developers-friendliness allowing for the rapid development of new algorithms. Tinker-HP offers the possibility to perform large scale simulations while keeping the Tinker reference double precision implementation dedicated to CPUs (Central Processing Unit) petascale architectures. The parallel scalability of the software has been demonstrated via benchmarks calculations. Overall, a several thousand-fold acceleration over a single-core computation is observed for the largest molecular systems, allowing therefore for reference long polarizable MD simulations on large molecular systems up to millions of atoms.

Despite this strong acceleration, and due to the development model of the Tinker software suite (now version 8 [2]), which favours new scientific development over optimization, no attempt has really been made yet to adapt the initial Tinker-HP code to a particular CPU architecture. Each execution of the code initially took little or no advantage of the underlying capabilities of the CPU it's running on. The existence of strong vector capabilities on modern Intel®architectures, and particularly the Intel®Advanced Vector Extensions 512 (Intel AVX-512) on Intel® Xeon® Scalable and Intel Xeon® Phi™ processors, leads us to change the overall design of the code, while trying to keep its simplicity and readability. The goal of this paper is double. First, it intends to propose a comprehensive living review dedicated to the capabilities and performances of Tinker-HP's main production methods (i.e. force fields) on Intel's architectures. Second, it is organized to present in a pedagogical way our code optimization efforts, giving a optimal strategy to vectorize Tinker-HP. Such practical case is rarely documented and we think it could be useful to many developers in the field.

The present version of the paper is then organized as follows. After reviewing the specificities of the latest Intel Xeon Scalable Processors (code-named Skylake) and particularly their Intel AVX-512 vector instructions set, we will give the general structure of the most computationally intensive FORTRAN subroutines in Tinker-HP, show their hotspots and propose a general strategy to vectorize the code with the Intel AVX-512 instruction set. We then give concrete examples of the vectorization process we follow. A performance comparisons between the released version and the vectorized version is then made, first for an execution on 1 core, to show brute acceleration of the code, and then in the context of a realistic parallel execution (with up to 16 000 cores). Finally, extended benchmarks on meaningful systems are provided with the AMOEBA polarizable force field [3–5] and also using the newly introduced initial implementation of classical force fields such as CHARMM [6], as an illustration of what we can obtain with any non-polarizable force field (AMBER,[7] OPLS-AA[8] etc...).

2 Intel Xeon Scalable processors

We are using in our study a system with a CPU from the Intel Xeon Scalable processors family (code-named Skylake). These processors feature up to 28 cores per processor with 2 hyper-threads per core for a total of up to 56 threads per processor. A new mesh interconnect reduces the latency of communication within cores and controllers in the processor. Each core is capable of issuing up to four instructions per cycle out-of-order. Up to two of them can be Intel AVX-512 instructions[9]. The Intel AVX-512 instruction set is a 512-bit SIMD instruction set that allows to perform computations with a single instruction using SIMD registers that contain eight double-precision (DP) or sixteen single-precision (SP) floating-point values as well as a variety of integer data sizes. The Intel AVX-512 instruction set also supports unaligned loads, fused-multiply and add, vector masking, shuffle and permutation instructions, histogram support, and hardware-accelerated transcendentals. Using all available cores and SIMD units is key to unlocking all the potential performance from these processors.

A significant change from its predecessor, the Intel Xeon processor v4, is the reorganization of the cache hierarchy to better balance the cache usage for server workloads. To achieve this, the L2 size has increased in size to 1MB and the last level cache (LLC) has been reduced in size (up to 38.5 MBs) but it is now a non-inclusive cache (meaning that data is not evicted from caches closer to the cores when evicted from the LLC).

The Intel Xeon Scalable processors provides two memory controllers with 3 memory channels each that support DDR4 up to 2600 MHz. This provides up to 123 GB/s of bandwidth to main memory for each socket. Three Intel®Ultra Path Interconnect links, each providing 10.4 GT/s, allow multiple processors to communicate to create bigger systems (e.g, dual-socket systems).

3 Considerations on vectorization

Efficient use of the SIMD units available in the processor is very important to achieve the best performances on modern (and probably future) processors. Most vector parallelism is extracted from loops. In this section, we outline some ideas that can help to get good performance from loop vectorization.

Modern compilers are able to auto-vectorize loops but

need to be able to determine that vectorization does not break any possible cross-iteration dependencies. This is not always possible due for example to variable aliasing or loop complexity[10]. Programmers can help compilers by rewriting their loops with idioms that compilers recognize and/or using directives that guide the vectorization process from the compiler.

Once a loop is vectorized, it is useful to consider if vector code generated for the loop is the most efficient possible. There are tools that can help on this assessment as well as providing feedback on how to fix it[11]. Common issues that are worth looked into are:

- **Unaligned loads or stores**. When data is not aligned to cache boundaries, some penalty will be incurred in load and store instructions compared to when data is well aligned. Therefore, it is recommended to align data to cache boundaries and also use the proper mechanisms to inform the compiler of this.
- **Loop prologues and remainders**. To provide better data alignment, the compiler might generate a different code for the first iterations of the loop to ensure the main vectorized part of the loop runs on aligned data. Similarly, if the compiler cannot deduce the number of iterations of a loop, it will generate a loop reminder that computes the last iterations of a loop that do not fill a vector register completely.
- **Unnecessary masking**. While the Intel AVX-512 instruction set supports vectorization of loops with conditional statements, supporting them requires the use of mask instructions and masked vector operations which reduces the vectorization efficiency. Therefore it is recommended to move as much as possible conditional statements out of vectorized loops. This might require to split the loop (with different code for those iterations where the branch was taken/not taken) or duplicate the loop (with different code for each branch of the conditional).
- **Non-unit strides**. Use of non-unit strides will most of the time force the compiler to generate gather and/or scatter instructions which are less efficient than regular loads and stores. This includes also accessing a field in a structure, as a non-unit stride will be needed to access the same field between consecutive elements of an array. This is why a Struct-of-Arrays (SoA) layout is preferred over a more conventional Array-of-Structs (AoS) layout for an efficient vectorization.
- **Indirect accesses**. Indexing one array with the values from another array will also result in generation of gather and/or scatter instructions. Therefore, this pattern should be avoided as much as possible.
- **Register pressure**. The number of vector registers is limited (e.g., Intel AVX-512 provides 32 vector registers). If the number of arrays that are used in a given loop plus temporary values that might be required for the operations of the loop exceeds the number of registers, the compiler will need to spill some of the arrays to the stack and restore them afterwards which reduces significantly the vectorization efficiency. To avoid this, it is better to use different loops for independent array operations rather a single big loop with all arrays inside.

4 Working environment and definitions

4.1 Definitions

In this paper, we will use two versions of Tinker-HP :

1. the Release Version 1.1, referred to as **Rel**.
2. the Vectorized Version 1.1v, referred to as **Vec**.

A third version, Release Version 1.2 (referred to as **Rel2**), is used in the Perspective subsection 8.3 to give a taste of the performance gain obtained with new algorithms. Vectorization of **Rel2** is in progress. It will be referred to as **Vec2**.

We ran Tinker-HP exclusively on *supercomputers* under UNIX/LINUX Operating System (OS). These machines aggregate hundreds or even thousands of interconnected systems called *computing nodes*, or simply *nodes*, each of which having tens of CPU cores and usually hundreds of gigabytes of memory. On UNIX/LINUX computers, the code is executed by a *process*, which uses memory and CPU resources managed by the OS.

What we called *the code* can be split in two parts :

1. the **User Code (UC)**, which comprises all the FORTRAN code. Here, it's the code for Tinker-HP and the 2DECOMP library.
2. the **Non–User Code (NUC)**, which comprises all the code executed by the process because of library calls from the **UC**, system calls done on behalf of the **UC** or code introduced implicitly by the compiler.

Of course, the way we write the **UC** (what library we used, how we setup our data,...) has an influence on what and how **NUC** is executed by the process. As we want to raise the performances, we have to take into account what **NUC** gets executed because of the **UC** code that we write.

We use the term *Molecular System* (**MS**) to denote all the physical systems we have simulated for this paper.

Note that the FORTRAN code listings shown in this paper have been taken *as is*, while all compilation reports and assembly code listings have been edited for publication purposes.

4.2 Compilation setup

We worked with the Intel® Parallel Studio XE 2018 development suite[12], containing the Intel® Fortran Compiler, the Intel® MPI Library, the Intel® Math Kernel Library (Intel MKL) with implementations of BLAS[13], LAPACK[14] and FFTW3[15] routines, and Intel®VTune™Amplifier for profiling and analysis.

The Tinker-HP sources are compiled with the flags shown in listing 1 where :

- `-xCORE-AVX512` flag forces the generation of binaries for the Intel Xeon Scalable processors.
- `-qopt-zmm-usage=high` flag instructs the compiler to use zmm (512 bits) registers as much as possible.
- `-align array64byte` instructs the compiler to align all static arrays to 64 bits memory address boundaries
- `-falign-functions=64` tells the compiler to align functions on 64 bits boundaries
- `-qopt-report-phase` and `-qopt-report=5` flags produce vectorization reports.
- `-S` flag produces assembly code listings.

```
FFLAGS = -O3 -xCORE-AVX512 -qopt-zmm-usage=high
         -no-ipo -no-prec-div -shared-intel
         -align array64byte -falign-functions=64
         -qopt-report-phase=vec -qopt-report=5 -S
         -qoverride-limits
```

Listing 1. Flags used for the compilation of Tinker-HP with Intel Fortran compiler.

Even if we do not use OpenMP*[16] in the dynamic simulation engine, other parts of the Tinkertools suite (Tinker 8 [2]) use OpenMP directives. So, object files are linked with flags shown in listing 2 where :

- `-mkl=sequential` flag tells the linker to use the sequential Intel MKL library, which is lighter and faster than the multi-threaded ones (e.g., OpenMP).
- `-qopenmp-stubs` flag enables compilation of OpenMP programs in sequential mode.

```
FFLAGS2 = -mkl=sequential -qopenmp-stubs
```

Listing 2. Flags used for the objects linking with Intel Fortran compiler.

4.3 Execution setup

For the performance tests, calculations have been performed on nodes running under the RedHat* Enterprise Linux* Server Release 7.4 operating system, with the following configuration :

- 2 × Intel Scalable Xeon 8168 processor – 2,7 GHz – 24 cores/processor
- 192 GB of DDR4 memory,
- InfiniBand* EDR interconnect.

We chose 8 **MS** from those studied in [1] with increasing sizes ranging from 9 737 to 3 484 755 atoms : the Ubiquitin protein, the prototypic Dihydrofolate Reductase (DHFR), the COX-2 dimer, the Satellite Tobacco Mosaic Virus (STMV), the Ribosome full structure in polarizable water, and three water boxes (Puddle, Pond and Lake).

The table 1 gives for each **MS** the name, the number of atoms and the number of cores used for the parallel calculations.

MS	Ubiquitin	DHFR	Puddle	COX-2
Atoms	9 737	23 558	96 000	174 219
CPU	480	680	1 440	2 400
CPU2		960		3 000
MS	Pond	Lake	STMV	Ribosome
Atoms	288 000	864 000	1 066 628	3 484 755
CPU	2 400	7 200	10 800	10 800
CPU2			16 200	16 200

Table 1. MS used for the performance measurements. The numbers of cores are taken from [1] for comparison. The CPU2 raw gives the number of cores which produced the best performance (See tables 5, 6 and 7). For the sequential performance measures, only one core was used.

All calculations have been made starting from an equilibrated **MS** with a timestep of 2*fs* and a RESPA integrator[17]. Times are in *seconds* for the one core tests, and timings in *ns/-day* for the parallel tests. The boost factors B are calculated as :

$$B = \frac{T_{\mathbf{Rel}}}{T_{\mathbf{Vec}}}$$

where $T_{\mathbf{Rel}}$ and $T_{\mathbf{Vec}}$ are times or timings for **Rel** and **Vec** respectively.

To get the profiles of **Rel** and **Vec**, we used DHFR **MS** with the AMOEBA polarizable force field and with the CHARMM classical force field (no polarization). Both simulations ran on one core and only 100 steps.

5 The Release version 1.1 of Tinker-HP

5.1 The polarizable AMOEBA force field

We focus here on the part of the code dedicated to the AMOEBA polarizable force field which is the most computationally challenging and gives a lower bound to Tinker-HP performances[3, 18]. AMOEBA has been shown to have a wide applicability for physical systems ranging from liquids to metals ions,[19, 20] including heavy ones,[21, 22] in solution and to proteins [4, 5] and to DNA/RNA[5]. It uses distributed atomic multipoles up to quadrupole moments and

a Thole/Applequist point dipole polarization model. Van der Waals interactions use the Halgren buffered 14–7 function. In this paper, we used the AMOEBA 18 protein parametrization [4, 5] coupled to the 2003 water model[3].

5.2 General Structure

Tinker-HP uses a 3D spatial decomposition to distribute atoms on the cores. Every process is assigned to a subsection of the simulation box and is responsible of updating the positions, speeds and accelerations of the atoms present in this subdomain at each timestep[1]. The most computationally intensive part of Tinker-HP is devoted to forces and electric fields calculations.

All the computation routines follow the same organization scheme :

- an external loop over all the atoms held by the core
- the selection of the neighbour sites using various criteria (e.g. cutoff distances, ...)
- a (very) big internal loop over the selected sites, where all quantities are computed.

In **Rel**, this internal loop computes all quantities for each atom-neighbour pair *on the fly*, with no attempt to pre-calculate or store intermediate quantities that can eventually be re-used. This results in a big harm on cache registers and processing units, and in a big use of memory-core transfer instructions. By contrast, there's almost no use of array (apart from indexing). Thus, the possibility to take advantage of the vector extension capabilities of the Intel AVX-512 instructions is very low.

5.3 Hotspots

The table 2 shows the profiling of **Rel**, running on 1 core for DHFR with AMOEBA (polarizable model) and with CHARMM (non-polarizable model). We give the module or routine, the real CPU time spent executing it, and the vector usage percentage. All routines are sorted with higher time-consuming first. We can see that **Rel** has two kinds of hotspots :

1. **NUC** hotspots : these are mainly due to libraries calls, system calls and memory management operations (initialization, copy, allocation and de-allocation).
2. Computational hotspots : these are mainly due to the computation of :
    - the matrix-vector product operation applied at each iteration of the polarization solver (`tmatxb_pme2`), which can be called up to 12 times at each step, depending on the convergence criterion
    - the dipole polarization energy and forces (`epolar1`)
    - the van der Waals energy and forces with Halgren buffered 14–7 function (`ehal1`)
    - the multipolar permanent electrostatic energy and forces (`empole1`)
    - the right hand size of the polarization equation (`efld0_direct2`)
    - the van der Waals energy and associated forces with Lennard-Jones 6-12 function (`elj1`)
    - the charge-charge interaction energy and associated forces (`echarge1`)

Module	CPU Time (s)	Vector usage %
NUC hostpots Total CPU time : **36.0896** s		
vmlinux	27.5005	100.00
libmkl_avx512.so	5.7625	100.00
libmpi.so.12.0	2.7144	0.00
libc-2.17.so	0.0862	0.00
libmkl_intel_lp64.so	0.0120	0.00
libiompstubs5.so	0.0090	0.00
libmpifort.so.12.0	0.0050	0.00
DHFR (AMOEBA, polarizable model) **Computational** hostpots Total CPU time : **278.9512** s (100 steps)		
tmatxb_pme2	100.9210	0.00
epolar1	52.7085	0.00
ehal1	52.4679	0.00
empole1	28.9127	0.00
image	25.2141	0.00
efld0_direct2	17.4910	0.00
torque	2.1355	0.00
DHFR (CHARMM, non-polarizable model) **Computational** hostpots Total CPU time : **24.7982*** s (100 steps)		
elj1*	15.3259	0.00
echarge1*	6.7309	0.00
image (1)	3.4130	0.00
image (2)*	2.7414	0.00

Table 2. Profiling of **Rel** using Intel VTune Amplifier. Simulations ran on one core and 100 steps. **MS** is DHFR with AMOEBA polarizable force field and with CHARMM force field (no polarization). Most important **NUC** and computational hostspots are shown in separate frames. **vmlinux** is the system kernel, performing memory operations and system calls. For CHARMM calculation, image is splitted in two parts. The vectorized routines will use image(2). So, only the starred lines are counted in the total CPU time for comparison with **Vec**.

Other widely used utility routines (`image` and `torque`) also appears.

In order to raise the performances of **Rel**, we needed to address these hotspots. Two observations have guided us :

1. **vmlinux** is taking almost as much CPU time as the multipole polarization energy and forces computation routine. That means the process makes many system calls and memory operations.
2. the vector usage percentage is strictly 0.00 for all the computation subroutines. It confirms that these routines only use scalar operations.

The first remark led us to investigate the library and system calls and, first and foremost, to work on the memory management of a process running Tinker-HP in order to reduce memory operations.

The second remark led us to rewrite the computation routines. As using vector operations means most of the time using loops on arrays, the *on the fly* method of computation in **Rel** must no longer be used.

6 Optimization strategy

6.1 Non-User Code hotspots

NUC hotspots come from libraries calls, system calls (file open, read or close, MPI function calls,...) and `memset`, `memcpy` and calls to `malloc()` or its derivatives that each process makes during its life.

6.1.1 Libraries and System calls

Libraries calls

The vast majority of libraries calls comes from the Intel MKL library, which actually does computing work. As using it wisely can give a significant speedup, we have to provide the right FORTRAN code (**UC**) to access the right vectorized functions of Intel MKL.

System calls

When running, Tinker-HP reads a few files at the very beginning and outputs a log file which contains the simulation results and, periodically, a file containing atoms coordinates. As these input/output operations are done by the MPI–rank–0 process only, the `open`, `read` and `close` system calls do not really stress the UNIX/LINUX system, even if the simulation runs on millions of atoms.

So, most of the system calls come from the MPI library and are due to two design choices in Tinker-HP :

1. As explained before, each Tinker-HP process hold a portion of the space (a domain) and maintains MPI communications with MPI processes that hold other domains nearby it, so that each process can exchange information with the others and track atoms coming in or out of its own domain. As the other processes can run on other nodes, there can be even more time spent in system calls because of network transmissions.
2. A memory region is shared between all MPI processes on a computing node, using the sharing capabilities of the MPI library. The access control of this region is implemented through system calls (semaphores, `flock()`...) to synchronize processes and guarantee non overlapping when writing data.

Minimizing the time spent in system calls is not so easy. We tried different distributions of the MPI processes over the nodes to favour local MPI communications, but that did not give convincing results. We also tried to improve the use of MPI framework by masking communications with computations. We used non blocking versions of `MPI_SEND` and `MPI_RECEIVE` functions, and did some calculations before calling the corresponding `MPI_WAIT`. The performance gain is not really noticeable for now. But improving this part of the code is a work in progress.

6.1.2 Memory management

The figure 1 gives a simple picture of the memory layout of a UNIX/LINUX process. The `text` zone contains the code, the `data` zone contains initialized data. The `bss` zone contains uninitialized fixed size data and has itself a fixed and limited size, set at compile time. The `heap` zone contains allocated data and can grow or shrink, subject to allocations or de-allocations. The `stack` zone contains a Last-In-First-Out structure, where values are pushed and pulled at each subroutine or function call or if the processor runs out of free registers.

Process memory setup

Historically, Tinker dynamically allocates and de-allocates all arrays it uses, because it was originally built to run on workstations with limited amount of memory and cores. These allocations are made by calls to the system `malloc()` group of functions. As a consequence, data are put in the `heap` section of the process, whose size is managed by the OS, allowing it to grow or shrink as needed.

As Tinker-HP is a pure MPI program which distributes data and can potentially run on hundreds of different nodes, each of them with gigabytes of memory, the problem of the memory consumption is not that important. On a computing node, each core (so, each MPI process) can easily have 2 or even 4 gigabytes of memory for its own use.

Still the size of some arrays are proportional to the size of the systems and therefore, the **MS**-size dependent data, declared when entering a subroutine, can be very large. In a

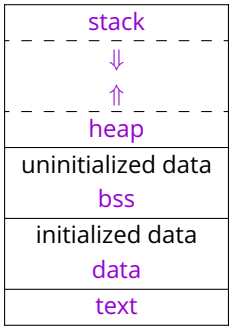

Figure 1. Memory layout of a running process. Arrows give the directions in which the zones expand.

normal run, each process maintains hundreds of numbers for each atom it holds. And we can have thousands of atoms held by each process and tens of MPI processes on one node. So, allocation and de-allocation of data for each process and in each subroutine constitutes a big stress for the OS.

Considering that the overall size of data held by one process is often under 2Gb, and that we maintain size derived constants throughout the program, we decided to remove all dynamic allocations in the vectorized routines, and declare all arrays with fixed sizes. That moves the data in the `bss` section of the process, which is lighter than `heap` for the OS to handle, lowering the stress on it.

Memset and memcpy

The execution cost of the `memset` and `memcpy` operations cannot be easily evaluated, as they come from the compiler libraries and are built upon the C library. But, because of their potential (big) effect on the performances (see table 2 and discussion on page 5), these operations have been extensively tracked.

Many of `memset` come from unnecessary zeroing and have been easily removed. Some of them come from the use of intrinsic FORTRAN90 functions, where the compiler creates temporary storage and introduces `memcpy` operations to work with it (for example, `PACK`). We tried to remove the use of intrinsic functions as much as possible. Some of the `memset` or `memcpy` operations also come from the way FORTRAN passes arrays to subroutines or functions (as a whole array, or as a slice). We avoided these operations wherever possible.

After this optimization, the real CPU time on one core for **NUC** hotspots can be shorter by up to 10%. But this depends a lot on the **MS** simulated and the activity of the UNIX/LINUX system outside of Tinker-HP.

6.2 Computational hotspots

The strategy we used can be developed in 4 guidelines :

1. **Rewrite all big internal loops**. As using vector operations means using arrays, big loops can be split in numerous short ones, each loop computing only one or a few quantities for all the involved atoms. This way, the quantities calculated can be kept in arrays and vector operations can be executed on them.
2. **Cope with the way the compiler works on loops**. As stated in section 3, when the compiler tries to vectorize a loop, it can generate 3 execution loops :
   - a **Peeled loop** (**P**–loop), to treat array elements up to the first aligned one.
   - a **Kernel loop** (**K**–loop), to treat the biggest possible number of array elements with vector operations.
   - a **Remainder loop** (**R**–loop), to treat elements that remain untreated by the previous loops.

   As the **K**–loops are the most effective and the fastest loops, we must eliminate as much **P**–loops and **R**–loops as possible. We'll show below what we did to achieve this goal.
3. **Use vectorized mathematical operations** as much as possible. This can be difficult sometimes, because each compiler or library implements them in its own way. For example, using the Intel Compiler, the `sqrt(X)` function is not vectorized. But the power function `**` is. So loops with `X**0.5` have a better vectorization score than loops with `sqrt(X)`. As the two functions can give slightly different numerical results, care must be taken to be sure to always keep the big accuracy needed by Tinker-HP.
4. **Have no dependency** between arrays in the loops, because the compiler will refuse to vectorize any loop where it cannot be sure that there is no dependency.

With that in mind, knowing that Intel AVX-512 vector registers can hold eight 8-bytes reals or sixteen 4-bytes integers, we should have a significant improvement of the speed if the number of neighbouring atoms is big enough to fill them. That's generally the case in Tinker-HP calculations, except for very in-homogeneous **MS**.

To summarize, filling in the 512 bits registers in an efficient way and using as much vector operations as possible in a loop need :

- **No dependency**, to be vectorized
- **Low number of arrays used**, to reduce the register pressure
- **Arrays as close as possible in memory**, to reduce cache miss and cost of memory operations
- **Data aligned on a suitable boundary**, to eliminate the **P**–loop
- **No subroutine calls, no un-vectorized math operations**, to get the best of the **K**–loop.

- **Loop count carefully chosen**, to eliminate the **R**–loop
- **No if-test**. If tests are mandatory (as in the selection process), they should be built in logical arrays before being used in the loop.

6.2.1 Dependency

Short loops calculate only one or a few unrelated quantities. They use the lower possible number of arrays. Thus, dependencies do not often occur. Where the non dependency cannot be automatically determined, we can easily see it and give directives to the compiler, or at worst rewrite the loop.

6.2.2 Data alignment

Historically, in Tinker, data were located in commons, that were themselves organized with scientific development in mind. Some compilers have options to align commons. But they may be inefficient if data are not correctly organized, with memory representation in mind.

We decided to replace commons with modules, which have many advantages :

- Arrays can be efficiently aligned using directives when declared
- Overall readability is better, due to modularity
- Code can be introduced in modules, so we can group operations and write them once and for all.

In all the modules, we used an `ATTRIBUTE ALIGN::64` directive for each array declaration. At the very beginning of this work, we used arrays like `pos(n,3)` to represent, for example, the three spatial coordinates of an atom. But, sometimes, we saw that the initial alignment of the first row of `pos` was not kept by the compiler for the following ones, preventing it from fully optimizing the code and forcing it to generate extra **P**–loops. All arrays are now mono-dimensional. The coordinates are represented by arrays like `Xpos(n)`, `Ypos(n)` and `Zpos(n)`. The three coordinates are treated in the same loop, with no dependency, allowing for vectorization.

6.2.3 Data layouts in memory

The figure 2 shows 3 different data layouts for arrays in memory.

- In the setup ①, no `ATTRIBUTE ALIGN::64` directive has been given. There is no memory loss, but the real*8 array is not on a 64bits boundary. During execution, elements in this array will not be aligned. If no `ASSUME_ALIGNED::64` directive is given, the compiler will generate **P**-loops. If an `ASSUME_ALIGNED::64` directive is given, no **P**-loop will be generated. But the process will pick up wrong real*8 numbers in the **K**–loop, and give wrong results, or even crash.

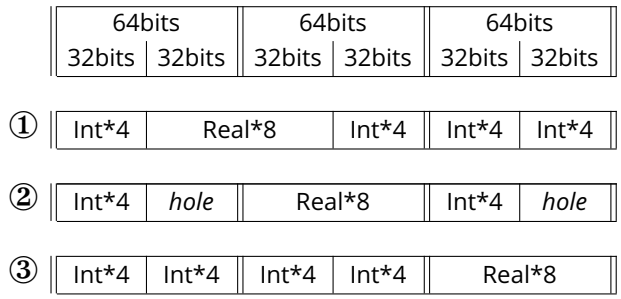

Figure 2. Schematic picture of 3 data layouts in memory. The double vertical separators show 64 bits boundary. The simple ones show 32 bits boundary.

- In the setup ②, all arrays are aligned on a 64 bits boundary with an `ATTRIBUTE ALIGN::64` directive. No **P**-loop will be generated. But there can be a memory hole, if the number of elements in the integer arrays is odd. When running, the process could have to do some jumps in memory to get the real*8 numbers, loosing time.
- In the setup ③, all arrays are aligned on 64bits with an `ATTRIBUTE ALIGN::64` directive. No **P**-loop will be generated. There's no memory hole, because the number of elements in the integer arrays is kept even.

```
!DIR$ ATTRIBUTES ALIGN:64::kglobvec1,kbisvec1
      integer kglobvec1(maxvlst),kbisvec1(maxvlst)
!DIR$ ATTRIBUTES ALIGN:64:: kvvec1,kvlocvec1
      integer kvvec1(maxvlst),kvlocvec1(maxvlst)
!DIR$ ATTRIBUTES ALIGN:64::rv7vec,rvvec2
      real*8 rv7vec(maxvlst),rvvec2(maxvlst)
!DIR$ ATTRIBUTES ALIGN:64::rikvec,rik2vec,rik3vec
      real*8 rikvec(maxvlst),rik2vec(maxvlst),rik3vec(
     maxvlst)
!DIR$ ATTRIBUTES ALIGN:64::rik4vec,rik5vec,rik6vec
      real*8 rik4vec(maxvlst),rik5vec(maxvlst),rik6vec(
     maxvlst)
!DIR$ ATTRIBUTES ALIGN:64::rik7vec,invrhovec,invtmpvec
      real*8 rik7vec(maxvlst),invrhovec(maxvlst),invtmpvec(
     maxvlst)
```

Listing 3. Typical array declarations in a module with alignment directives. Integer arrays precede real*8 arrays. Arrays are ordered as per their utilization wherever possible.

We decided to implement the setup ③, which represents the best trade-off between speed and memory consumption. So, we ended up with the typical array declarations in a module (here, `MOD_vec_vdw`) shown in listing 3, where :

- All 4-bytes integer arrays are listed before 8-bytes real ones, and each parametric dimension is a multiple of 64, eliminating the holes in the data layout and ensuring correct alignment in any cases. For example, in the listing 3, the parameter `maxvlst`, which represents the maximum number of neighbours in van der Waals pair list, is set to 2560.

- Arrays used in a loop are listed close to each others in modules and in the same order of utilization as much as possible, to reduce the time for memory operations.

For arrays declared in the subroutines (outside of modules), with shapes depending on the size of the **MS**, we calculated the next multiple of 16 bigger than the size, and used it as the dimension of the array.

Although the setup ③ implies an over-consumption of memory for arrays with **MS**–dependent sizes, this is almost not noticeable, because we add at worst 15 elements to the size, which goes usually from thousands to millions in Tinker-HP.

Almost all **P**–loops have been removed this way.

6.2.4 Loop counts

As we have many short loops in each subroutine, we really must carefully choose the loop counts. The number of sites being dependent of the size of the **MS** we work on, we cannot impose a fixed value. To be sure to completely fill the 512 bits registers at each loop, we decided to maintain two working loop counts :

1. a **Real loop** count, multiple of 8 for loops on real*8 arrays ($8 * 8 = 64$ bytes, so 512 bits).
2. an **Integer loop** count, multiple of 16 for loops on integer arrays ($16 * 4 = 64$ bytes, so 512 bits)

As an example, if `nnvlst` is the number of sites we work on, these working loop counts are computed as in listing 4.

```
nvloop8  = (int(nnvlst /  8) + 1) *  8
nvloop16 = (int(nnvlst / 16) + 1) * 16
nvloop8  = merge(nnvlst, nvloop8  , mod(nnvlst,  8) .eq. 0)
nvloop16 = merge(nnvlst, nvloop16 , mod(nnvlst, 16) .eq. 0)
```

Listing 4. Calculation of the working loop counts for real and integer operations

Since `nnvlst` can already be a multiple of 8 or 16, we should use the `mod` and `merge` constructs to get the smallest loop count. We used `nvloop8` for `real*8` operations, and `nvloop16` for `integer` operations. `real*8` arrays are loaded in registers by chunks of 8 elements, and `integer` arrays by chunks of 16 elements. We eliminated almost all the **R**–loops this way.

The flaw of this method is that, when `nnvlst` is very low, we do an overwork. However, the number of neighbours is generally big enough (typically between 200 and 2000), so these extra iterations are not too expensive. Furthermore, they are executed as **K**–loops, where maximum vectorization is in effect. We just have to remember that we have now extra (useless) calculated values and find a way to drop them in our calculations.

6.2.5 Design of loops

The use of short loops allows the programmer to better understand the code. That's exactly the same for the compiler ! So, loops in **Vec** are :

- **simple**. We use a small number of arrays (a maximum of 8 seems to be a good start). Due to the intrinsic complexity of the mathematical model, we use many temporary arrays.
- **short**. The loops contains 10 instructions at most, and 3 or 4 most of the time.
- **mostly if-test free**. Most of the time, an if-test in a loop prevents the compiler from vectorizing. For the loops in the neighbours selection process, which cannot be if-test free by nature, tests are built as logical array masks before being used. This way, loops using these masks can easily be vectorized.

6.2.6 Editing considerations

Big editing effort may seem to be useless at first glance. But we found that, most of the time, it was far easier to understand and vectorize loops once they are really readable. Also, modifications and debugging can be faster.

7 Typical vectorized loops in Tinker-HP

We have 2 kinds of vectorized loops in Tinker-HP :

- **selection loops** that select sites using various cutoffs and work on integers or logicals
- **compute loops** that compute quantities and work on reals.

We will show each of them in details, and give some insights on how we have improved the vectorization. The typical loops have been extracted from `ehal1vec`, which use the module `MOD_vec_vdw` shown in listing 3.

7.1 Typical selection loop

We built a selection mask `mask1` with the appropriate test, using `nvloop16` here, as we work on integers (listing 5). We first tell the compiler to assume the loop count value is a multiple of 16, so that it does not generate any **R**–loop. We really have to be sure of that, otherwise the process will pick up numbers from other arrays in the memory and wrong results or bad crashes will occur.

```
!DIR$ ASSUME (MOD(nvloop16,16).eq.0)
 do k = 1, nvloop16
    mask1(k)= (kvlocvec(k) /= 0).and.(kbisvec (k) <= nbloc)
&                               .and.(kvlocvec(k) <= nbloc)
 enddo
```

Listing 5. Loop creating a logical mask.

The vectorization report on listing 6 shows that the loop is perfectly vectorized.

We then applied the mask on the set of atoms we worked on to select those of interest. In FORTRAN, we have an intrinsic function `PACK` that does exactly this. So, in a first attempt, we wrote the selection loop as shown in the listing 7.

Unfortunately, because of the `PACK` function, which does an implicit loop over all the elements of the array it works on, each line was seen as an independent loop by the compiler, and optimization was made on that line only.

```
LOOP BEGIN at (284,10)
   reference mask1(k) has aligned access
   vector length 16
   normalized vectorization overhead 0.660
   LOOP WAS VECTORIZED
   unmasked aligned unit stride loads: 3
   unmasked aligned unit stride stores: 1
   --- begin vector cost summary ---
   scalar cost: 44
   vector cost: 3.120
   estimated potential speedup: 13.840
   --- end vector cost summary ---
LOOP END
```

Listing 6. Vectorization report for the mask creation. Notice that the speedup reported is not in time of execution, but in number of operations.

```
      nnvlst1    = count(mask1)
      kglobvec1  = pack(kglobvec,mask1)
      kbisvec1   = pack(kbisvec ,mask1)
      kvvec1     = pack(kvvec   ,mask1)
      kvlocvec1  = pack(kvlocvec,mask1)
```

Listing 7. First attempt with the pack function.

The corresponding FORTRAN compiler report on listing 8 shows that a vectorized loop is generated for the `count` line, with a vector cost of 0.810. Then, the first `PACK` line generates 3 loops :

- one over the 2-bytes logical array `mask1` (vector length 32) with a vector cost of 1.250.
- one over the 4-bytes integer array `kglobvec` (vector length 16) with a vector cost of 1.000.
- one reduced to a `memset` or `memcpy` for the affectation of `kglobvec1`

The total vector cost is 2.250 for one `PACK` operation. We also have 3 loads and 1 store for each `PACK` line.

For this selection loop, we obtained a total vector cost of $0.810 + 4 * 2.250 = 9.810$, plus 4 `memset` or `memcpy`, and a total of 13 loads and 4 stores. We cannot easily know the final cost of this selection loop, because, as stated before, the implementation of the 4 memory operations depends on the compiler.

To get a controlled and constant vector cost, whichever compiler we use, we decided to get rid of the `PACK` function.

```
LOOP BEGIN at (309,22)
   vector length 16
   unroll factor set to 2
   normalized vectorization overhead 1.192
   LOOP WAS VECTORIZED
   unmasked aligned unit stride loads: 1
   --- begin vector cost summary ---
   scalar cost: 11
   vector cost: 0.810
   estimated potential speedup: 13.330
   --- end vector cost summary ---
LOOP END

LOOP BEGIN at (311,22)
   vector length 32
   normalized vectorization overhead 0.800
   LOOP WAS VECTORIZED
   unmasked aligned unit stride loads: 1
   --- begin vector cost summary ---
   scalar cost: 11
   vector cost: 1.250
   estimated potential speedup: 8.710
   --- end vector cost summary ---
LOOP END

LOOP BEGIN at (311,22)
   reference kglobvec(:) has aligned access
   vector length 16
   normalized vectorization overhead 0.188
   LOOP WAS VECTORIZED
   unmasked aligned unit stride loads: 2
   masked unaligned unit stride stores: 1
   --- begin vector cost summary ---
   scalar cost: 12
   vector cost: 1.000
   estimated potential speedup: 11.980
   --- end vector cost summary ---
   vector compress: 1
LOOP END

LOOP BEGIN at (311,10)
   loop was not vectorized: loop was transformed to
  memset or memcpy
LOOP END
```

Listing 8. Vectorization report for the selection loop (pack version)

After all, packing data is just a matter of selecting array elements and putting them contiguously in a new array. So, we ended up with a functionally equivalent loop depicted in listing 9.

Although there is a test in this loop, the corresponding FORTRAN compiler report (listing 10) clearly shows that :

- The loop is vectorized
- Every reference is aligned, so are the loads. The stores cannot be aligned, because of the packing.
- The vector length is 16 which means 16 integers will be picked up in each operation
- The potential speedup is more than 7. This is very good in the presence of a test.
- 4 vector compress instructions are generated, which correspond to the 4 affectations.

```
        kk = 0
!DIR$ ASSUME (MOD(nvloop16,16).eq.0)
        do k = 1, nvloop16
           if (mask1(k)) then
              kk = kk + 1
              kglobvec1(kk) = kglobvec(k)
              kbisvec1 (kk) = kbisvec (k)
              kvvec1   (kk) = kvvec   (k)
              kvlocvec1(kk) = kvlocvec(k)
           endif
        enddo
        nnvlst1 = kk
```

Listing 9. Final selection loop with no PACK function.

The last remark is very interesting : the Intel compiler was able to recognize this construct as a packing loop, and implemented it directly with `vpcompressd` instructions, which are Intel AVX-512 pack instructions at the machine code level.

A look to the assembly code in listing 11 confirms that the `vpcompressd` instructions operate on `zmm` pure 512-bit vector registers.

```
   LOOP BEGIN at (298,10)
      reference kglobvec(k) has aligned access
      reference kbisvec (k) has aligned access
      reference kvvec   (k) has aligned access
      reference kvlocvec(k) has aligned access
      vector length 16
      normalized vectorization overhead 0.300
      LOOP WAS VECTORIZED
      unmasked aligned unit stride loads: 5
      masked unaligned unit stride stores: 4
      --- begin vector cost summary ---
      scalar cost: 18
      vector cost: 2.500
      estimated potential speedup: 7.090
      --- end vector cost summary ---
      vector compress: 4
   LOOP END
```

Listing 10. Vectorization report for the selection loop.

```
vpcompressd %zmm3, vec_mp_kglobvec1_(,%rcx,4){%k1}
vpcompressd %zmm4, vec_mp_kbisvec1_(,%rcx,4){%k1}
vpcompressd %zmm5, vec_vdw_mp_kvvec1_(,%rcx,4){%k1}
addl        %edx, %r14d
incl        %edx
movslq      %edx, %rdx
vpcompressd %zmm6, -4+vec_vdw_mp_kvlocvec1_(,%rdx,4){%k1}
```

Listing 11. Typical selection loop assembly code extract showing the vpcompressd instructions.

We obtained a vector cost of only 2.500 and no `memset` or `memcpy`. That's 4 times smaller than the `PACK` version, and much more if we count the `memset` or `memcpy` operations. The number of loads is also reduced by a factor of 3. This version is really faster than the first one.

7.2 Typical compute loop

The calculation loops follow the scheme we have described above. They are short, simple and easy to read. The listing 12 shows a typical compute loop.

We first tell the compiler to assume the loop count value is a multiple of 8 (we work on reals here). We use the arrays in the order they were declared in the module (see listing 3). There is no dependency, since all arrays are independent and used after being completely set. The compiler can easily optimize the loads and stores.

```
!DIR$ ASSUME (MOD(nvloop8,8).eq.0)
do k = 1, nvloop8
   rv7vec (k)   = rvvec2 (k)  ** 7
   rik3vec(k)   = rik2vec(k)  * rikvec(k)
   rik4vec(k)   = rik3vec(k)  * rikvec(k)
   rik5vec(k)   = rik4vec(k)  * rikvec(k)
   rik6vec(k)   = rik5vec(k)  * rikvec(k)
   rik7vec(k)   = rik6vec(k)  * rikvec(k)
   invrhovec(k) = (rik7vec(k) + ghal * rv7vec(k)) ** - one
   invtmpvec(k) = (rikvec (k) + dhal * rvvec2(k)) ** - one
enddo
```

Listing 12. A typical compute loop. Notice that there are 8 instructions and 11 different array references.

As all arrays here are aligned, no **P**–loop are generated by the compiler. Because of the loop count, no **R**–loop are generated either.

```
   LOOP BEGIN at (417,10)
      reference rvvec2   (k) has aligned access
      reference rv7vec   (k) has aligned access
      reference rik3vec  (k) has aligned access
      reference rik2vec  (k) has aligned access
      reference rikvec   (k) has aligned access
      reference rik4vec  (k) has aligned access
      reference rik3vec  (k) has aligned access
      reference rikvec   (k) has aligned access
      reference rik5vec  (k) has aligned access
      reference rik4vec  (k) has aligned access
      reference rikvec   (k) has aligned access
      reference rik6vec  (k) has aligned access
      reference rik5vec  (k) has aligned access
      reference rikvec   (k) has aligned access
      reference rik7vec  (k) has aligned access
      reference rik6vec  (k) has aligned access
      reference rikvec   (k) has aligned access
      reference rik7vec  (k) has aligned access
      reference rv7vec   (k) has aligned access
      reference invrhovec(k) has aligned access
      reference rikvec   (k) has aligned access
      reference rvvec2   (k) has aligned access
      reference invtmpvec(k) has aligned access
      vector length 8
      normalized vectorization overhead 0.049
      LOOP WAS VECTORIZED
      unmasked aligned unit stride loads: 14
      unmasked aligned unit stride stores: 8
      --- begin vector cost summary ---
      scalar cost: 272
      vector cost: 25.750
      estimated potential speedup: 10.490
      vectorized math library calls: 2
      --- end vector cost summary ---
   LOOP END
```

Listing 13. Vectorization report for the compute loop.

We can see from the corresponding FORTRAN compiler report in the listing 13 that :

- The loop is vectorized
- Every reference is aligned, so are the loads and stores.
- The vector length is 8 which means 8 numbers will be picked up in each operation
- The potential speedup is around 10.5.
- 2 vectorized math library calls are made for the 2 ** function.

```
jle       ..B2.324
cmpl      $8, %r13d
jl        ..B2.460
movl      %r13d, %edx
xorl      %eax, %eax
andl      $-8, %edx
movslq    %edx, %rdx
vbroadcastsd vdwpot_mp_ghal_(%rip), %zmm17
vbroadcastsd vdwpot_mp_dhal_(%rip), %zmm16
vmovups   -816(%rbp), %zmm20
movl      %r14d, -456(%rbp)
movq      %rdx, %r14
movq      %r12, -152(%rbp)
movq      %rax, %r12
# MAIN VECTOR TYPE: 64-bits floating point
vmovups   vec_vdw_mp_rikvec_(,%r13,8), %zmm19
vmovups   vec_vdw_mp_rvvec2_(,%r13,8), %zmm18
vmulpd    vec_vdw_mp_rik2vec_(,%r13,8), %zmm19, %zmm5
vmulpd    %zmm18, %zmm18, %zmm2
vmulpd    %zmm19, %zmm5, %zmm6
vmulpd    %zmm2, %zmm2, %zmm3
vmulpd    %zmm18, %zmm2, %zmm4
vmovupd   %zmm5, vec_vdw_mp_rik3vec_(,%r13,8)
vmulpd    %zmm19, %zmm6, %zmm7
vmulpd    %zmm4, %zmm3, %zmm0
vmovupd   %zmm6, vec_vdw_mp_rik4vec_(,%r13,8)
vmulpd    %zmm19, %zmm7, %zmm8
vmovupd   %zmm0, vec_vdw_mp_rv7vec_(,%r13,8)
vmovupd   %zmm7, vec_vdw_mp_rik5vec_(,%r13,8)
vmulpd    %zmm19, %zmm8, %zmm9
vmovupd   %zmm8, vec_vdw_mp_rik6vec_(,%r13,8)
vfmadd213pd %zmm9, %zmm17, %zmm0
vmovupd   %zmm9, vec_vdw_mp_rik7vec_(,%r13,8)
vmovaps   %zmm20, %zmm1
call      *__svml_pown8_z0@GOTPCREL(%rip)
vfmadd231pd %zmm16, %zmm18, %zmm19
vmovaps   %zmm20, %zmm1
vmovupd   %zmm0, vec_vdw_mp_invrhovec_(,%r12,8)
vmovaps   %zmm19, %zmm0
call      *__svml_pown8_z0@GOTPCREL(%rip)
vmovupd   %zmm0, vec_vdw_mp_invtmpvec_(,%r12,8)
addq      $8, %r12
cmpq      %r14, %r12
jb        ..B2.322
```

Listing 14. Typical calculation loop assembly code showing vector only operations.

A look to the assembly code in listing 14 shows that all multiplications are done with vector operations `vmulpd` and `vfmadd213pd` and `vfmadd231pd`, which are fused multiply-add operations. These vector instructions operate on `zmm` registers. We can also see the two calls to the vectorized version of the ** function so we are fully using Intel AVX-512 capabilities.

If ever we had used a division, instead of `**- one`, we would have got :

```
      estimated potential speedup: 4.610
      divides: 2
      --- end vector cost summary ---
```

Listing 15. Excerpt of a vectorization report for the compute loop with division.

The estimated potential speedup in this case is less than half the previous one. And the utilization of the vector units is not so optimal.

So, a careful reading of the vectorization report is always necessary to ensure the best choices have been made.

7.2.1 Final profile for **Vec**.

The tables 3 shows the profile and the boost factors between **Rel** and **Vec** for the final vectorized routines.

NUC hotspots

The Real CPU Time is almost the same as for the **Rel** version. We can see a reduction of about 10% for the real CPU time of **vmlinux**. The `libmkl_vml_avx512.so` shared library has appeared, because we use vectorized mathematical functions and replaced all calls to the complementary error function `erfc`, which was in the sources of **Rel**, by calls to `vderfc`, which is a vectorized implementation in Intel MKL library.

Computational hotspots

The vector usage percentage varies between 64% and 100%, and the boost factors are between 1.60 and more than 5.10. The real CPU time has shrunk from 278.95*s* to 136.46*s* for AMOEBA calculation, and from 24.8*s* to 13.0*s* for CHARMM calculation. The vectorized parts of code are now about 2 times faster.

8 Performance on Intel Scalable Processors

8.1 Sequential performance

We have evaluated the brute performance boost of Intel AVX-512 by running calculations on only one core from a dedicated node. In this situation, we can easily measure the time of execution of each interesting part of the code, with limited perturbation coming from the MPI part of the code or the presence of other processes.

We chose to measure 3 execution times :

1. **time_vdw**, which is the time taken by Van der Waals calculations. Depending on the setup, we used ehal1(vec) or elj1(vec).

Module	CPU Time (s)	Vector Usage %	Boost factor
NUC hostpots Total CPU time : **37.3438** s			
vmlinux	25.4116	100.00	
libmkl_avx512.so	6.1404	100.00	
libmpi.so.12.0	2.7094	0.00	
libmkl_vml_avx512.so	2.6733	100.00	
libc-2.17.so	0.0703	0.00	
libmkl_intel_lp64.so	0.3208	0.00	
libmpifort.so.12.0	0.0110	0.00	
libiompstubs5.so	0.0070	0.00	
DHFR (AMOEBA, polarizable) **Computational** hostpots Total CPU time : **136.4569** s (100 steps)			**2.0442**
tmatxb_pme2vec	62.9675	**63.90**	1.6027
epolar1vec	29.2285	**94.90**	1.8033
ehal1vec	19.9918	**67.90**	2.6245
empole1vec	11.7175	**90.20**	2.4675
efld0_direct2vec	6.9914	**82.60**	2.5018
imagevec	4.9416	**100.00**	5.1024
torquevec2	0.6186	**85.70**	3.4521
DHFR (CHARMM, no polarization) **Computational** hostpots Total CPU time : **13.0355*** s (100 steps)			**1.9076**
elj1vec*	8.2493	**64.60**	1.8578
echarge1vec*	3.6786	**90.90**	1.8297
image (1)	3.4130	0.00	1.0000
imagevec*	1.1076	**100.00**	2.4751

Table 3. Profiling of **Vec** using Intel VTune Amplifier. Simulations ran on one core and 100 steps. **MS** is DHFR with AMOEBA polarizable force field and with CHARMM force field (no polarization). For the vectorized routines, the Vector Usage percentages go from 63.9 to 100%. Only the starred lines are counted in the total CPU time for DHFR with CHARMM.

2. **time_elec**, which is the time taken by electrostatic calculations. Depending on the setup, we used echarge1(vec) (charges) or empole1(vec) (multipoles).
3. **time_polar**, which is the time taken by polarization calculation and by far the biggest. We used epolar1(vec).

In this case, the boost is always a tradeoff between feeding the CPU with enough numbers, which goes better with the size of the **MS**, and minimizing the exchanges between memory and cores, which goes worse with the size.

For every **MS**, we ran 10 steps of calculations using the polarizable AMOEBA force field. We took the average value of each times, removing the lowest and the biggest. Results are given in the table 4.

MS	Ubiquitin		
	Time **Rel** (s)	Time **Vec** (s)	Boost
time_vdw	0.0964	0.0567	1.7002
time_elec	0.1352	0.0967	1.3981
time_polar	1.2326	0.8758	1.4062
MS	DHFR		
	Time **Rel** (s)	Time **Vec** (s)	Boost
time_vdw	0.2359	0.1453	1.6235
time_elec	0.2823	0.2012	1.4031
time_polar	2.6051	1.8181	1.4329
MS	COX-2		
	Time **Rel** (s)	Time **Vec** (s)	Boost
time_vdw	1.8906	1.1362	1.6639
time_elec	2.3816	1.7398	1.3689
time_polar	22.2782	15.6152	1.4267
MS	STMV		
	Time **Rel** (s)	Time **Vec** (s)	Boost
time_vdw	1.9878	1.2260	1.6214
time_elec	3.8826	2.9314	1.3245
time_polar	64.2167	45.4406	1.4132

Table 4. 1 step measured times of execution and boost factors for different test **MS** using **Rel** or **Vec**. Simulations ran on 1 core. Values are averaged over 10 steps.

Performance boost factors are always very good, even when the size of the **MS** is very big (more than 1 million atoms on one core for STMV !). The boosts we obtained are very significant and justify the important vectorization efforts we made to get them.

But the real gain should be estimated in a more *realistic* situation, where Tinker-HP is running in parallel. In this case, there could be from 8 to 48 processes running on one node, each competing for resources, and up to 340 nodes involved, multiplying MPI communications.

8.2 Best parallel performances

8.2.1 Polarizable force field : AMOEBA

We focus here on the absolute timings improvements over previous publications. As vectorization did not affect the scaling of the methods, interested readers can refer to the Tinker-HP software publication for detailed analysis of the scalability [1]. Here, we ran calculations on 2 000 steps (4ps), with the core setups shown in table 1. The best performance was taken as the average of the 20 performance evaluations made by Tinker-HP during the run, after having removed the first, middle and last values, which are lower because Tinker-HP writes intermediate files at these steps.

Results given in table 5 shows a boost factor between 1.45 and 1.59 in parallel mode. The boost increases with the size

AMOEBA polarizable Force Field			
MS	**Rel** timings (ns/day)	**Vec** timings (ns/day)	Boost factors
Ubiquitin	11.6875	16.9779	1.4526
DHFR (CPU)	9.1725	13.3312	1.4533
DHFR (CPU2)	9.4761	14.6054	1.5413
Puddle	3.5421	5.2417	1.4798
COX-2 (CPU)	1.9608	2.9343	1.4965
COX-2 (CPU2)	2.0137	3.1514	1.5650
Pond	1.7921	2.7620	1.5412
Lake	0.7066	1.1025	1.5602
STMV (CPU)	0.4995	0.7921	1.5858
STMV (CPU2)	0.5152	0.8140	1.5799
Ribosome (CPU)	0.2295	0.3420	1.4901
Ribosome (CPU2)	0.2368	0.3527	1.4894

Table 5. Best production timings and boost factors for the different **MS** using **Rel** or **Vec**. For DHFR, COX-2, STMV and Ribosome, optimal results with CPU2 setup are also shown (see table 1).

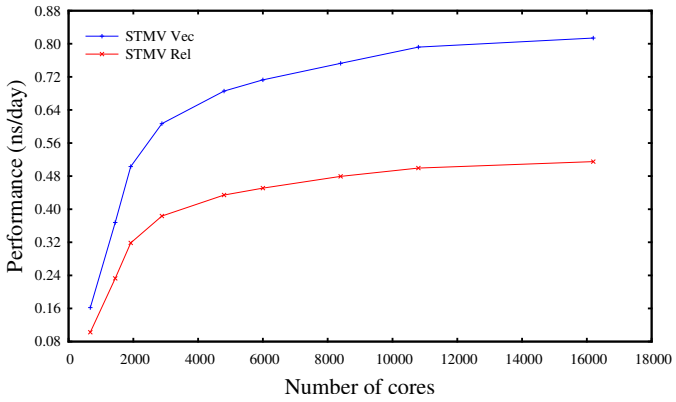

Figure 3. Performance gain for the STMV using **Rel** or **Vec**. The boost factor decreases from 1.59 to 1.57 when increasing the number of cores.

of the **MS**, indicating a better overall utilization of the vector registers. When **MS** tend to be very big, other phenomena (MPI memory contention, network communications, ...) lower the boost factors. We are still able to get little gains with CPU2 sets, because most of the supplementary cores use vectorized routines. Anyway, these results are very encouraging, knowing that not all the code has been optimized.

We pushed forward and tried simulations on STMV and Ribosome with up to 16200 cores (CPU2 set). The figures 3 and 4 give the performances obtained for **Rel** and **Vec** when increasing the number of cores.

The boost factors remain relatively constant for these two **MS**. With very large number of cores (and very large number of nodes), both **Rel** and **Vec** speeds are bounded by MPI communications and memory operations.

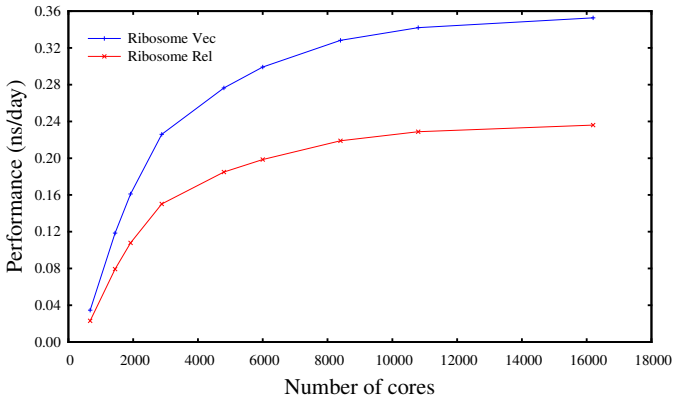

Figure 4. Performance gain for the ribosome using **Rel** or **Vec**. The boost factor decreases from 1.51 to 1.49 when increasing the number of cores.

8.2.2 Non-polarizable force field : CHARMM

Tinker-HP is yet not optimized for these kind of force fields as no specific "modern algorithmics" is present. Indeed, the initial implementation is mainly a massively parallel version of the initial Tinker that was initially aimed at performing comparisons for Steered Molecular dynamics between polarizable and non-polarizable approaches.[23] Also, we used for the tests a very conservative molecular dynamics setup where bonds are not rigidified, reciprocal space computations are done at each timestep, etc... Such setup is chosen in order to provide reference numbers but actual timings could be accelerate in many ways. Thus, we know that we need much more cores to get results comparable to those of other prominent codes[24–26]. Nevertheless, we decided to make timings, firstly to get an idea of the boost the vectorization can provide in this case and, secondly, to know if we can still benefit from the scalability of the code, which is one of its greatest strengths. We used the same **MS** and the same CPU sets, limiting us to a maximum of 2 400 cores, since using more cores would not make any sense and knowing that these setups may not be optimal for this other type of computations (i.e. as they were chosen for AMOEBA).

Vectorization boost

The table 6 shows the timings we obtained for **Rel** and **Vec**.

Overall, the speedup factor in using non-polarizable force fields is found to be between 3 and 4. The boost factors are lower than for AMOEBA, mainly because the vectorized part of the code which is really executed is itself smaller. They show the same behaviour than for AMOEBA when the size of the **MS** increases, with a peak value reached for smaller systems (around 200 000 atoms). Beyond this size, the non-vectorized code become the limiting speed factor.

CHARMM non polarizable Force Field			
MS	**Rel** timings (ns/day)	**Vec** timings (ns/day)	Boost factors
Ubiquitin	39.3068	48.8269	1.2422
DHFR (CPU)	24.2333	31.7408	1.3098
DHFR (CPU2)	26.4805	34.8272	1.3152
Puddle	9.4749	12.8026	1.3512
COX-2	8.1411	11.3459	1.3936
Pond	5.1579	6.8394	1.3260

Table 6. Best production timings and boost factors for different **MS** using **Rel** or **Vec** with CHARMM force field. For DHFR, optimal results with CPU2 setup are also shown (see table 1).

Scalability

We tested the scalability of the code with 3 **MS** : Ubiquitin, DHFR and COX-2. As for AMOEBA, we ran 2000 steps of calculation with increasing number of cores, and took the average performance given by the code. The figures 5, 6 and 7 show the performances obtained for **Rel** and **Vec** when increasing the number of cores.

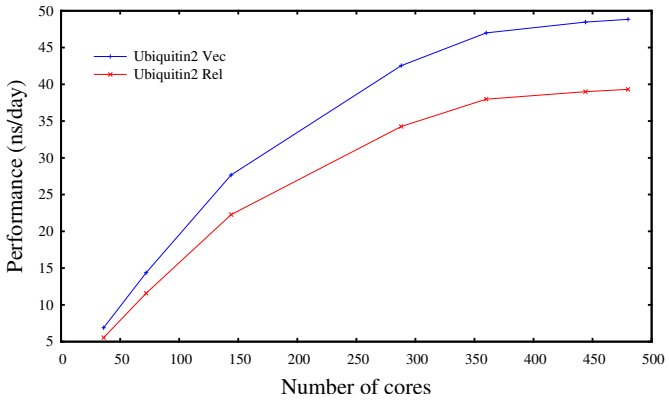

Figure 5. Performance gain with CHARMM forces field for the Ubiquitin using **Rel** or **Vec**. The boost factor remains constant when increasing the number of cores.

Whatever **MS** we simulate, the scalability is still very good. The boost factor remains almost constant for the 2 smaller **MS** (Ubiquitin and DHFR). For the COX-2, the boost factor decrease from 1.41 to 1.39 when increasing the number of cores, because, with 2 400 cores, communications tends to lower the benefit of the vectorization. In practice, this part of the code is a first step towards an efficient engine for non-polarizable MD but some work is still required and is in progress to obtain better performances with an updated module to come.

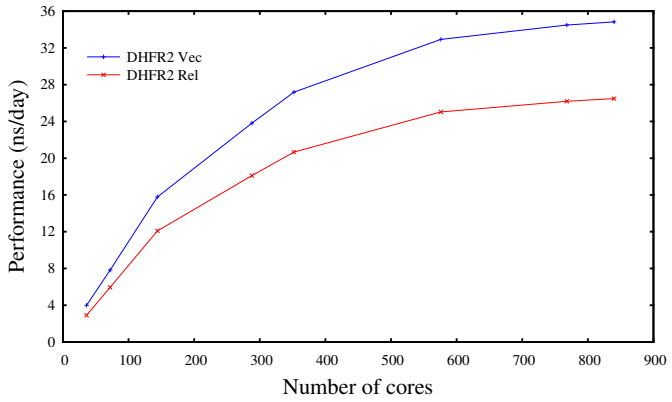

Figure 6. Performance gain with CHARMM forces field for the DHFR using **Rel** or **Vec**. The boost factor remains almost constant when increasing the number of cores.

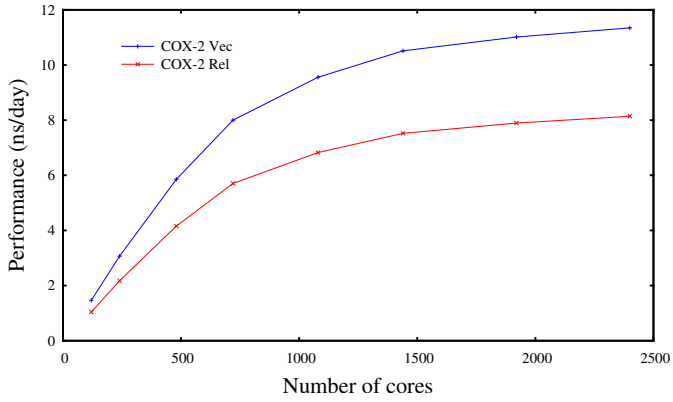

Figure 7. Performance gain with CHARMM forces field for the COX-2 using **Rel** or **Vec**. The boost factor slightly decrease when increasing the number of cores.

8.3 Perspectives on Tinker-HP 1.2 performance

This section intends to give a taste of the incoming performance gain that will appear in the Tinker-HP Release 1.2 version (**Rel2**). Indeed, despite being not fully vectorized yet, this major update proposes significant algorithmic speedups. For now, we can point out that a strong performance gain without accuracy loss can be observed by using Tinker-HP coupled to the new multi-timestep BAOAB-RESPA1 integrator[27] with hydrogen mass repartitioning. This newly introduced integrator splits the energy terms in 3 levels evaluated at different timesteps: the bonded terms are evaluated every 1fs, the non-bonded terms (including polarization) are split into short and long range, the short-range being evaluated every 10/3 fs and the long range every 10fs. Furthermore, short-range polarization is treated with the non-iterative TCG-1 (Truncated Conjugate Gradient) solver[28, 29] and the outer-level uses the Divide-and-Conquer Jacobi Iterations (DC-JI) [30]

approach, offering a net global acceleration about 4.91 time compared to standard 1fs/Beeman/ASPC (7 without ASPC) simulations without loss of accuracy enabling a good evaluation of properties such as free energies[27].

Preliminary results (where not all routines are yet vectorized) are reported in Table 7. We intend to review the full 1.2 vectorized version in an update of this living review.

AMOEBA polarizable Force Field			
MS	**Rel2** (ns/day)	**Rel2-multi** (ns/day)	**Vec2-multi** (ns/day)
Ubiquitin	11.6875	28.28	40.32
DHFR (CPU)	9.1725	22.20	32.26
DHFR (CPU2)	9.4761	22.93	35.33
Puddle	3.5421	8.57	12.68
COX-2 (CPU)	1.9608	4.74	7.09
COX-2 (CPU2)	2.0137	4.87	7.65
Pond	1.7921	4.34	6.69
Lake	0.7066	1.70	2.65
STMV (CPU)	0.4995	1.21	1.92
STMV (CPU2)	0.5152	1.25	1.97
Ribosome (CPU)	0.2295	0.55	0.82
Ribosome (CPU2)	0.2368	0.57	0.85

Table 7. Best production timings for the different **MS** using **Rel2**, **Rel2-multi** (multi-timestep) and **Vec2-multi** (multi-timestep). For DHFR, COX-2, STMV and Ribosome, optimal results with CPU2 setup are also shown (see table 1).

Finally, beside the focus on the AMOEBA polarizable force field, timings will be given for other polarizable models as well as on classical force fields (CHARMM, AMBER, OPLS etc...). For now, despite the non-optimization and the absence of use of lower precision of this part of the code, more than a 4-time speedup of the values reported in Table 7 give an initial idea of the reasonable code performances for non-polarizable simulations.

9 Conclusion

This work has been a fundamental step in many ways.

First, it demonstrates that new HPC architectures can offer large acceleration to existing massively parallel code like Tinker-HP. A brute performance boost between 1.32 and 1.70 can be achieved on computationally intensive parts of the code, leading to an overall acceleration factor between 1.45 and 1.59 for AMOEBA (1.24 and 1.40 for CHARMM) in realistic conditions, even for the simulation of molecular systems with millions of atoms. Considering that the most costly calculations must be done with a simulation time of a few microseconds, such a gain in speed is undoubtedly a big progress.

Second, it shows that every gain in speed is not just a matter of raising the frequency of the CPU or buying more powerful computers. Such a large gain involves a close cooperation between the computational chemists, who write the code, and the HPC specialists, who know how the CPU and the system software work. To get this acceleration, we had to dig into the pieces of code that were the most CPU consuming and to rewrite them almost completely, with simplicity and efficiency in mind. But it was worth the effort. Furthermore, considering the past trend of past CPUs, we consider that vectorization will also play an important role in future architectures.

Third, it gives us a strategy and some methods to further improve the code. It can serve as a solid starting point for the future. We are now able to more easily adapt Tinker-HP to new underlying hardware or software technologies. That would allow us to make the best of them.

Of course, optimization is far from finished as some parts of the code are not yet vectorized (for example the reciprocal space computations of permanent electrostatics and polarization) and other sources of speedups exist and will be investigated. In particular, we have to review how we can improve the creation of the neighbour lists, how we can have a faster indexing of all the atoms (sorting indexes could be a solution) and how we can better mask MPI communications with computations. Decreasing precision is also possible in specific cases to gain performances while retaining accuracy. This paper never ceases to be updated as we will accumulate new data on Github until a new version of this living paper is pushed to review. The next iteration of this paper will also incorporate results on next generation of Intel Xeon Scalable processors (codenamed Cascade Lake) and intend to evolve towards the adaptation of the Tinker-HP code to any of the future architectures proposed by Intel. Future work will focus on the algorithmic boosting of our initial implementation of classical non-polarizable force fields as the next iteration of the paper will propose more detailed benchmarks about incoming new polarizable approaches including SIBFA [31, 32] and major evolutions of AMOEBA such as AMOEBA+[33] and HIPPO[34, 35].

10 Acknowledgments

We want to give a special thank to David Guibert and Cedric Bourasset (Center for Excellence in Parallel Programming, Atos-Bull) for initial discussions.

LHJ wishes to thank Prof Esmaïl Alikhani (Sorbonne Université) for sharing his 2x18 cores Skylake computer in the early phase of this work.

LHJ also wants to thank Bernard Voillequin for fruitful discussions at the very beginning of this work. B. Voillequin is not a scientist, not even an engineer. But he's a man of good

sense and good will. That's invaluable.

For a more detailed description of contributions from the community and others, see the GitHub issue tracking and changelog at https://github.com/TinkerTools/Tinker-HP/tree/master/LiveCOMS.

11 Funding Information

The authors acknowledge the generous computing time support provided by GENCI (Grand Equipement National de Calcul Intensif, France) through the Grand Challenge and grant no A0050707671 computing allocated on the Irène Joliot-Curie (TGCC, Bruyère le Châtel, France), Occigen and Frioul machines (CINES, Montpellier, France) supercomputers. The authors also thank the TACC (Texas Advanced Computer Center) for computational resources on the Stampede-2 machine. JWP and PR thank the support by National Institutes of Health (R01GM106137 and R01GM114237).

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

For more information go to www.intel.com/benchmarks.

Performance results are based on testing as of March 14th 2019 and may not reflect all publicly available security updates. See configuration in the paper for details. No product or component can be absolutely secure.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at [intel.com].

Intel, the Intel logo, Xeon, VTune and Xeon Phi are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Author Information

ORCID:

Luc-Henri Jolly	:	0000-0003-3220-5759
Louis Lagardère	:	0000-0002-7251-0910
Jay W. Ponder	:	0000-0001-5450-9230
Pengyu Ren	:	0000-0002-5613-1910
Jean-Philip Piquemal	:	0000-0001-6615-9426

References

[1] **Lagardère L**, Jolly LH, Lipparini F, Aviat F, Stamm B, Jing ZF, Harger M, Torabifard H, Cisneros GA, Schnieders MJ, Gresh N, Maday Y, Ren PY, Ponder JW, Piquemal JP. "Tinker-HP: a Massively Parallel Molecular Dynamics Package for Multiscale Simulations of Large Complex Systems with Advanced Polarizable Force Fields". Chemical Science. 2018; 9:956–972. https://doi.org/10.1039/c7sc04531j.

[2] **Rackers JA**, Wang Z, Lu C, Laury ML, Lagardère L, Schnieders MJ, Piquemal JP, Ren PY, Ponder JW. "Tinker 8: Software Tools for Molecular Design". J Chem Theory Comput. 2018; 14(10):5273–5289. https://doi.org/10.1021/acs.jctc.8b00529, pMID: 30176213.

[3] **Ren PY**, Ponder JW. "Polarizable Atomic Multipole Water Model for Molecular Mechanics Simulation". J Phys Chem. 2003; 107(24):5933–5947. https://doi.org/10.1021/jp027815+.

[4] **Shi Y**, Xia Z, Zhang J, Best R, Ponder JW, Ren P. The Polarizable Atomic Multipole-Based AMOEBA Force Field for Proteins. J Chem Theory Comput. 2013; 9(9):4046–4063. https://doi.org/10.1021/ct4003702.

[5] **Zhang C**, Lu C, Jing Z, Wu C, Piquemal JP, Ponder JW, Ren PY. "AMOEBA Polarizable Atomic Multipole Force Field for Nucleic Acids". J Chem Theory Comput. 2018; 14:2084–2108. https://doi.org/10.1021/acs.jctc.7b01169.

[6] **MacKerell AD**, Bashford D, Bellott M, Dunbrack RL, Evanseck JD, Field MJ, Fischer S, Gao J, Guo H, Ha S, Joseph-McCarthy D, Kuchnir L, Kuczera K, Lau FTK, Mattos C, Michnick S, Ngo T, Nguyen DT, Prodhom B, Reiher WE, et al. All-Atom Empirical Potential for Molecular Modeling and Dynamics Studies of Proteins. The Journal of Physical Chemistry B. 1998; 102(18):3586–3616. https://doi.org/10.1021/jp973084f, pMID: 24889800.

[7] **Wang J**, Wolf RM, Caldwell JW, Kollman PA, Case DA. Development and testing of a general amber force field. Journal of Computational Chemistry. 2004; 25(9):1157–1174. https://doi.org/10.1002/jcc.20035.

[8] **Jorgensen WL**, Maxwell DS, Tirado-Rives J. "Development and Testing of the OPLS All-Atom Force Field on Conformational Energetics and Properties of Organic Liquids". J Am Chem Soc. 1996; 117:11225–11236. https://doi.org/10.1021/ja9621760.

[9] **Intel Corp** , Architecture - Instruction Set Extensions Programming Reference; 2014.

[10] **Maleki S**, Gao Y, Garzaran MJ, Wong T, Padua DA. An evaluation of vectorizing compilers. In: *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11)* IEEE Computer Society; 2011. p. 372–382. https://doi.org/10.1109/PACT.2011.68.

[11] **Intel Corp** , Vectorization Advisor; 2019. https://software.intel.com/en-us/advisor/features/vectorization.

[12] **Intel Corp** , Intel Parallel Studio XE; 2019. https://software.intel.com/en-us/parallel-studio-xe.

[13] **Blackford LS**, Petitet A, Pozo R, Remington K, Whaley RC, Demmel J, Dongarra J, Duff I, Hammarling S, Henry G, et al. An updated set of basic linear algebra subprograms (BLAS). ACM Transactions on Mathematical Software. 2002; 28(2):135–151. https://doi.org/10.1145/567806.567807.

[14] **Anderson E**, Bai Z, Dongarra J, Greenbaum A, McKenney A, Du Croz J, Hammarling S, Demmel J, Bischof C, Sorensen D. LAPACK: A Portable Linear Algebra Library for High-performance Computers. In: *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing* Supercomputing '90, Los Alamitos, CA, USA: IEEE Computer Society Press; 1990. p. 2–11. https://doi.org/10.1109/SUPERC.1990.129995.

[15] **Frigo M**, Johnson SG. The design and implementation of FFTW3. Proceedings of the IEEE. 2005; 93(2):216–231. https://doi.org/10.1109/JPROC.2004.840301.

[16] **OpenMP ARB**, OpenMP Application Programming Interface Version 5.0; 2018. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[17] **Tuckerman M**, Berne BJ, Martyna GJ. Reversible multiple time scale molecular dynamics. J Chem Phys. 1992; 97(3):1990–2001. https://doi.org/10.1063/1.463137.

[18] **Ponder JW**, Wu C, Ren PY, Pande VS, Chodera JD, Schnieders MJ, Haque I, Mobley DL, Lambrecht DS, DiStasio RAJ, Head-Gordon M, Clark GNI, Johnson ME, Head-Gordon T. "Current Status of the AMOEBA Polarizable Force Field". J Phys Chem B. 2007; 114(8):2549–64. https://doi.org/10.1021/jp910674d.

[19] **Piquemal JP**, Perera L, Cisneros GA, Ren P, Pedersen LG, Darden TA. Towards accurate solvation dynamics of divalent cations in water using the polarizable amoeba force field: From energetics to structure. The Journal of Chemical Physics. 2006; 125(5):054511. https://doi.org/10.1063/1.2234774.

[20] **Wu JC**, Piquemal JP, Chaudret R, Reinhardt P, Ren P. Polarizable Molecular Dynamics Simulation of Zn(II) in Water Using the AMOEBA Force Field. J Chem Theory Comput. 2010; 6:2059–2070. https://doi.org/10.1021/ct100091j.

[21] **Marjolin A**, Gourlaouen C, Clavaguéra C, Ren PY, Wu JC, Gresh N, Dognon JP, Piquemal JP. Toward accurate solvation dynamics of lanthanides and actinides in water using polarizable force fields: from gas-phase energetics to hydration free energies. Theoretical Chemistry Accounts. 2012; 131(4):1198. https://doi.org/10.1007/s00214-012-1198-7.

[22] **Marjolin A**, Gourlaouen C, Clavaguéra C, Ren PY, Piquemal JP, Dognon JP. Hydration gibbs free energies of open and closed shell trivalent lanthanide and actinide cations from polarizable molecular dynamics. Journal of Molecular Modeling. 2014; 20(10):2471. https://doi.org/10.1007/s00894-014-2471-6.

[23] **Célerse F**, Lagardère L, Derat E, Piquemal JP. Massively Parallel Implementation of Steered Molecular Dynamics in Tinker-HP: Comparisons of Polarizable and Non-Polarizable Simulations of Realistic Systems. Journal of Chemical Theory and Computation. 0; 0(0):null. https://doi.org/10.1021/acs.jctc.9b00199, pMID: 31059250.

[24] **Abraham MJ**, Murtola T, Schulz R, PÃ¡ll S, Smith JC, Hess B, Lindahl E. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. SoftwareX. 2015; 1-2:19 – 25. https://doi.org/https://doi.org/10.1016/j.softx.2015.06.001.

[25] **Phillips JC**, Braun R, Wang W, Gumbart J, Tajkhorshid E, Villa E, Chipot C, Skeel RD, KalÃ© L, Schulten K. Scalable molecular dynamics with NAMD. Journal of Computational Chemistry. 2005; 26(16):1781–1802. https://doi.org/10.1002/jcc.20289.

[26] **Kobayashi C**, Jung J, Matsunaga Y, Mori T, Ando T, Tamura K, Kamiya M, Sugita Y. GENESIS 1.1: A hybrid-parallel molecular dynamics simulator with enhanced sampling algorithms on multiple computational platforms. Journal of Computational Chemistry. 2017; 38(25):2193–2206. https://doi.org/10.1002/jcc.24874.

[27] **Lagardère L**, Aviat F, Piquemal JP. Pushing the Limits of Multiple-Time-Step Strategies for Polarizable Point Dipole Molecular Dynamics. The Journal of Physical Chemistry Letters. 2019; 10:2593–2599. https://doi.org/10.1021/acs.jpclett.9b00901.

[28] **Aviat F**, Levitt A, Maday Y, Stamm B, Ren PY, Ponder JW, Lagardère L, Piquemal JP. "Truncated Conjugate Gradient (TCG): an optimal strategy for the analytical evaluation of the many-body polarization energy and forces in molecular simulations". J Chem Theory Comput. 2017; 13:180–190. https://doi.org/10.1021/acs.jctc.6b00981.

[29] **Aviat F**, Lagardère L, Piquemal JP. "The Truncated Conjugate Gradient (TCG), a Non-iterative/Fixed-cost Strategy for Computing Polarization in Molecular Dynamics: Fast Evaluation of Analytical Forces". J Chem Phys. 2017; 147:161724. https://doi.org/10.1063/1.4985911.

[30] **Nocito D**, Beran GJO. "Massively Parallel Implementation of Divide-and-Conquer Jacobi Iterations Using Particle-Mesh Ewald for Force Field Polarization". Journal of Chemical Theory and Computation. 2018; 14(7):3633–3642. https://doi.org/10.1021/acs.jctc.8b00328, pMID: 29847125.

[31] **Gresh N**, Cisneros GA, Darden TA, Piquemal JP. Anisotropic, polarizable molecular mechanics studies of inter-, intra-molecular interactions, and ligand-macromolecule complexes. A bottom-up strategy. J Chem Theory Comput. 2007; 3(6):1960–1986. https://doi.org/10.1021/ct700134r.

[32] **Piquemal JP**, Chevreau H, Gresh N. Toward a Separate Reproduction of the Contributions to the Hartree-Fock and DFT Intermolecular Interaction Energies by Polarizable Molecular Mechanics with the SIBFA Potential. Journal of Chemical Theory and Computation. 2007; 3(3):824–837. https://doi.org/10.1021/ct7000182, pMID: 26627402.

[33] **Liu C**, Piquemal JP, Ren P. AMOEBA+ Classical Potential for Modeling Molecular Interactions. J Chem Theory Comput. 2019; https://doi.org/10.1021/acs.jctc.9b00261.

[34] **Rackers JA**, Wang Q, Liu C, Piquemal JP, Ren P, Ponder JW. An optimized charge penetration model for use with the AMOEBA force field. Phys Chem Chem Phys. 2017; 19:276–291. https://doi.org/10.1039/C6CP06017J.

[35] **Rackers JA**, Ponder JW. Classical Pauli repulsion: An anisotropic, atomic multipole model. The Journal of Chemical Physics. 2019; 150(8):084104. https://doi.org/10.1063/1.5081060.

List of Figures

List of Tables

Listings