Parallel dual tree traversal on multi-core and many-core architectures for astrophysical N-body simulations

Benoit Lange, Pierre Fortin

Parallel dual tree traversal on multi-core and many-core architectures for astrophysical N-body simulations

Benoit Lange * [abc] and Pierre Fortin[bc]

[a]Sorbonne Universités, UPMC Univ Paris 06,
Institut Calcul et Simulation, F-75005 Paris, France
[b]Sorbonne Universités, UPMC Univ Paris 06,
UMR 7606, LIP6, F-75005, Paris, France
[c]CNRS, UMR 7606, LIP6, F-75005, Paris, France
Email: {benoit.lange,pierre.fortin}@lip6.fr

May 30, 2014

Abstract

In astrophysical N-body simulations, Dehnen's algorithm, implemented in the serial *falcON* code and based on a dual tree traversal, is faster than serial Barnes-Hut tree-codes, but outperformed by parallel CPU and GPU tree-codes. In this paper, we present a parallel dual tree traversal, implemented in the *pfalcON* code, targeting multi-core CPUs and many-core architectures (Xeon Phi). We focus here on both performance and portability, while preserving Dehnen's original algorithm. We first use task parallelism, with either OpenMP or Intel TBB, for the dual tree traversal. We then rely on the SPMD (single-program, multiple-data) model for the SIMD vectorization of the near field part thanks to the Intel SPMD Program Compiler. We compare the *pfalcON* performance to related work, and finally obtain performance results that match one of the best current tree-code implementations on GPU.

Keywords: dual tree traversal, task parallelism, SIMD, SPMD model, N-body problem

1 Introduction

The N-body problem describes the computation of all pairwise interactions among N bodies (or particles). Once computed, the corresponding forces are used to update the body positions and velocities for the next time-step. In astrophysics, such N-body simulations are essential and widely used for galactic dynamics studies. The gravitational force computation is the most time-consuming part and limits in practice the number of bodies, which is currently much smaller than the number of stars in a real galaxy.

The direct computation of all pairwise interactions among N bodies leads to a prohibitive $\mathcal{O}(N^2)$ runtime complexity. Hierarchical methods [2,5] have therefore been introduced to reduce

*This work undertaken (partially) in the CALSIMLAB framework is supported by the public grant ANR-11-LABX-0037-01 of the French National Research Agency (ANR) as part of the "Investissements dAvenir" program (ANR-11-IDEX-0004-02).

this runtime complexity: thanks to an octree data structure, the force field is decomposed in a near field part, directly computed, and a far field part approximated with various expansions. In astrophysics, the Barnes-Hut tree-code is one of the most used algorithms for serial and parallel CPU executions (see for example *treecode1* in NEMO[1] and GADGET-2 [12]). Recently, parallel implementations on GPUs (Graphics Processing Units) [3, 4] have also been developed which outperform multi-core CPUs.

Dehnen's algorithm [6], implemented in the serial *falcON* code (Force ALgorithm with Complexity $\mathcal{O}(N)$), is one order of magnitude faster than serial executions of Barnes-Hut tree-codes [6, 9], mainly thanks to its dual tree traversal (DTT). But parallel tree-codes implementations, on one or two multi-core CPUs or on one GPU, then manage to outperform *falcON* [9]. The parallelization of *falcON* is therefore crucial to exploit its algorithmic asset on current parallel architectures. But contrary to tree-codes algorithms, this DTT does not exhibit natural parallelism.

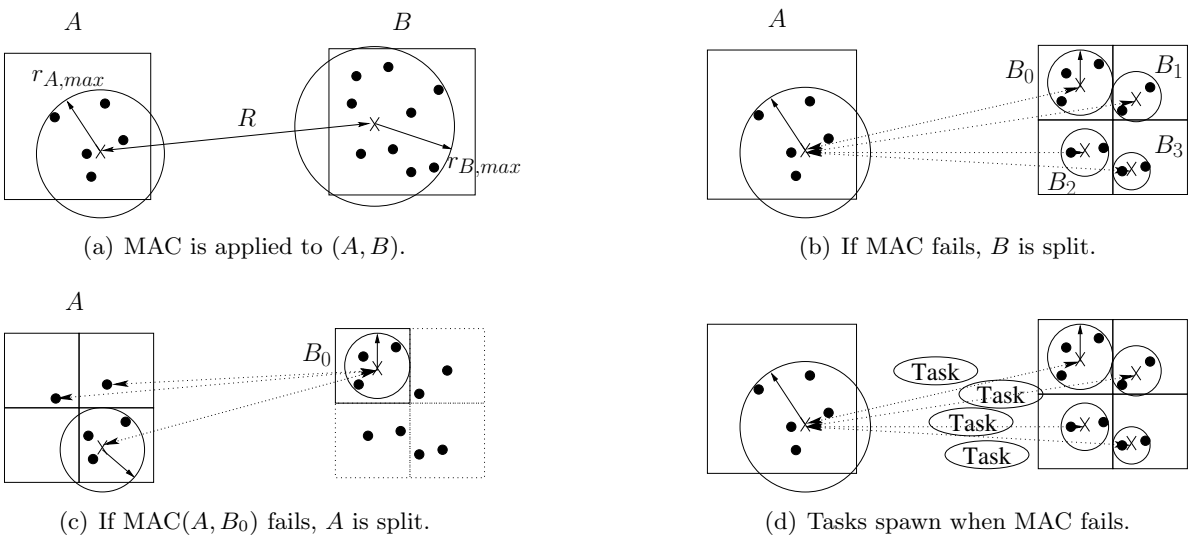
In this paper, we present a parallel dual tree traversal, implemented in the *pfalcON* (*parallel falcON*) code, that efficiently exploits two levels of parallelism on one single shared-memory node (we do not consider distributed-memory parallelism here). We first target multi-core parallelism on CPUs whose number of cores is constantly increasing, as well as on new many-core architectures like the Intel Xeon Phi whose compute power is similar to high end GPUs. We also target SIMD parallelism because of its increasing importance in the overall CPU performance: 128-bit SSE, 256-bit AVX, and 512-bit Xeon Phi vector units.

Contributions. Our contributions are thus two-fold. Firstly, we use task parallelism for the DTT on both multi-core CPUs and on the Xeon Phi. This requires a recursive formulation of the *falcON* code, as well as adequate atomic operations and memory barriers in order to obtain an efficient implementation. We detail how this can be achieved for both OpenMP tasks and Intel TBB (Threading Building Blocks) tasks, and how we manage to preserve Dehnen's original algorithm in the parallel tree traversal. Secondly, we use Intel SPMD Program Compiler (`ispc`) and its SPMD (single-program, multiple-data) model for the SIMD vectorization of the direct computation required for the near field part. We show that such approach enables us to have one single portable source code for this direct computation which is very efficient on both SSE and AVX, as well as on Xeon Phi vector instructions. Best performance is here obtained via a hybrid strategy that efficiently combines scalar and vector code. In the end, we show performance results that match the GPU `Bonsai` code which is currently one of the fastest GPU tree-codes [3].

Related work. An MPI parallelization of Dehnen's algorithm has been briefly presented in [10], but is based on a complete rewriting of the algorithm in Fortran 90, not on the highly optimized C++ *falcON* code. Recently, the `exaFMM-dev` software has included an implementation of Dehnen's algorithm that also uses task parallelism for the dual tree traversal [13, 15], but in a different way that requires the rewriting of this traversal. As for SIMD programming, `exaFMM-dev` uses C++ template metaprogramming for the hand-tuned kernel of the direct computation part. In the following, we will thus highlight the differences between *pfalcON* and `exaFMM-dev` and compare their performance.

In the rest of this paper, Sect. 2 describes N-body algorithms, especially Dehnen's algorithm. In Sect. 3, we detail how we have used task parallelism and SPMD programming in the *pfalcON* code. Section 4 presents performance results and comparisons with other codes. Finally, concluding remarks will be presented in Sect. 5.

[1] A Stellar Dynamics Toolbox: `http://bima.astro.umd.edu/nemo`



(a) MAC is applied to (A, B).

(b) If MAC fails, B is split.

(c) If MAC(A, B_0) fails, A is split.

(d) Tasks spawn when MAC fails.

Figure 1: Dual tree traversal in Dehnen's algorithm.

2 N-body algorithms

We focus here on galactic simulations and on hierarchical N-body algorithms. We do not consider other N-body algorithms such as those based on Particle-Mesh methods [12] for example. In the hierarchical methods, the 3D particle space is hierarchically decomposed by means of an octree. This octree is built by inserting particles one by one and by subdividing octree leafs containing more than a given maximum number of particles, denoted by *Ncrit*.

The Barnes-Hut tree-code algorithm [2] computes the gravitational forces among N particles with a $\mathcal{O}(N \ln N)$ runtime complexity thanks to monopole (and possibly quadrupole) moments. For each target body, the octree is here recursively traversed and "body-cell" or "body-body" interactions are evaluated depending on the multipole acceptance criterion (MAC): $D/r < \theta$, where D denotes the octree cell side length, r is the distance from the target body to the cell center of mass, and θ is an input parameter that balances accuracy and computation cost. Such expansions are well-suited for the relatively low precisions required in astrophysical N-body simulations, where a relative force error of few 10^{-3} is usually adequate. The loop on the target bodies is parallel which enables CPU parallel implementations with multi-threading and/or with MPI [12, 14]. Recently, this inherent parallelism has been efficiently exploited to develop GPU implementations that run entirely on the GPU [3, 4]. For example, the `Bonsai` code, which relies on monopole and quadrupole moments and on a specific MAC, enables speedups around 20 on GPU compared to a multi-core CPU implementation [3].

Dehnen's algorithm [6] can be considered as a nontraditional fast multipole method [5], specific to the relatively low precisions required in astrophysics. This $\mathcal{O}(N)$ algorithm indeed relies on "cell-cell" interactions. This requires specific, low precision local expansions based on cartesian Taylor expansions, and a specific MAC that can balance (along with the expansion order, which is fixed to 3) the accuracy and the computation cost. This MAC is defined for two cells (A, B) (see Fig. 1(a)) as: $\frac{r_{A,max}+r_{B,max}}{R} < \theta$, where $r_{C,max}$ denotes an upper limit for the distance of any body within the node C from its center of mass [6]. Once the octree has been built and the multipole moments have been calculated, the interactions are computed in two steps.

The first step (*interaction phase*) is presented in Algorithm 1 and relies on the dual tree traversal (DTT) presented in Figs. 1(a),1(b),1(c). If the MAC succeeds between two cells (A, B),

Algorithm 1 Interact(cell A, cell B)

```
1: if (MAC fails between A and B) then
2:     if (A = B) then
3:         for all pairs {a, aa} of children of A do
4:             Interact(a, aa)
5:         end for
6:     else if r_{A,max} > r_{B,max} then
7:         for all children a of A do
8:             Interact(a, B)
9:         end for
10:    else
11:        for all children b of B do
12:            Interact(A, b)
13:        end for
14:    end if
15: else
16:    Approximate interaction (A, B)
17: end if
```

Algorithm 2 Evaluate(cell A, local expansion L_0)

```
1: translate L_0 to center of mass of A
2: add L_0 to L_A (local expansion of A)
3: if A is a leaf then
4:     for all bodies b in A do
5:         evaluate L_A at position of b
6:         add it to potential and force of b
7:     end for
8: else
9:     for all children c of A do
10:        Evaluate(c, L_A)
11:    end for
12: end if
```

their interactions can be approximated: both local expansions of A and B are updated thanks to the mutuality of gravity (see Fig. 1(a)). If the MAC fails, the larger cell (B here) is split and the MAC is applied between A and all the children of B (see Fig. 1(b), with 8 children in 3D). This is applied recursively, and A can then be split when the MAC fails with A as the larger cell (see Fig. 1(c)). This thus leads to a dual recursive traversal of the octree. When the MAC fails for two octree leafs, or when the number of particles is too low (depending on empirical thresholds [6]), the direct computation is used instead of the expansions. Thanks to this DTT, Dehnen's algorithm consistently uses the mutuality of gravity to halve the computation cost in the near field part as well as in the far field part. This DTT also enables to better preserve the total momentum than tree-codes [6].

After the interaction phase, the *evaluation phase* is recursively used to evaluate the local expansion of each cell for each body within this cell. This evaluation phase is presented in Algorithm 2 and first called on the root cell with an empty local expansion.

All these features have been implemented in the *falcON*[2] code (Force ALgorithm with Complexity $\mathcal{O}(N)$) which offers $\mathcal{O}(N)$ computation times one order of magnitude smaller than serial executions of Barnes-Hut tree-codes [6,9]. Moreover, these computation times are much less sensitive to the distribution of particles: this is very important for astrophysical simulations where the particle distributions representing galaxies or groups of galaxies are highly non-uniform.

The interaction and evaluation steps correspond together to the most time consuming part. The octree construction has a non-negligible computation time, but it does not have to be performed at every time-step. Moreover, the interaction step represents around 95% of the total time for both the interaction and evaluation steps, which makes it crucial for the overall performance of *falcON*. In the following, we will thus see how the interaction and evaluation steps of Dehnen's algorithm can be efficiently parallelized in a new *pfalcON* code.

3 *pfalcON*: a parallel *falcON*

The dual tree traversal of the interaction step was described as a recursive algorithm in [6], but was implemented using a stack-based approach in the *falcON* code. In practice when an interaction fails the MAC, it is pushed in one of the available stacks according to its type ("cell-cell", "body-cell"...). The stacks are then regularly popped in a specific order. When using

[2] Available in `http://carma.astro.umd.edu/nemo/`, version 3.6. We use here the *gyrfalcON* full-fledged N-body code (GalaxY simulatoR using falcON).

tasks to process interactions, the task runtime will have to store the tasks (i.e. the interactions) in memory. The original stacks in *falcON* then become redundant. We have thus removed these stacks and rewritten *falcON* as a recursive code (more convenient for task parallelism), where recursive calls process interactions in the same order as in *falcON*. This new code will be hereafter referred to as *rfalcON*.

Besides, local Taylor expansions are allocated on the fly during the dual tree traversal in the original *falcON* code. This enables to save memory by allocating these expansions only when they are effectively required for each cell. In parallel executions, concurrent memory allocations have to be serialized at the system level, which can become a performance bottleneck. In *rfalcON* we thus allocate these expansions for each non-empty cell during the step which computes multipole moments. This implies some memory overhead, which is not problematic since current N-body simulations on one single node are more limited by the compute power than by the available memory.

With such features, *rfalcON* is slightly faster than *falcON* (around 8%) for the interaction step.

3.1 Task parallelism for the dual tree traversal

On multi-core CPUs, loop-based parallelism (like in OpenMP) is suitable for tree-codes, but clearly not here for the DTT of the *rfalcON* interaction phase since there is no explicit parallel loop. Task parallelism, firstly introduced in Cilk and now available in OpenMP (since version 3.0) and in Intel TBB, is here much more suitable for such recursive algorithm. Tasks are specified in the source code by the programmer, and then managed during the execution by a runtime which dynamically schedules these tasks on the available threads. Such dynamic load balancing is especially useful in astrophysical N-body simulations where the particle distributions, hence the computation loads, are highly non-uniform.

In *pfalcON*, each time an interaction fails the MAC, we thus simply create one task for each of the (up to) eight interactions involving the children of the larger cell: see Fig. 1(d). However, due to the consistent use of the mutuality of gravity in Dehnen's algorithm, a task updates both cells A and B (either local expansions or particles) when the interaction is effectively computed. Hence different tasks can update the same cells concurrently which requires synchronization among the tasks to avoid conflicts. We need here the lightest synchronization mechanism to have the smallest overhead on the parallel execution. That is why we use here atomic operations and memory barriers on one specific flag per cell to indicate if the cell is already being updated or not.

More precisely, we use here one bit (the Most Significant Bit - MSB) in one 32-bit integer variable (named `val` in the *falcON* code). Such variable is stored in each octree cell to describe various features of this cell, and only 25 bits are currently used. When a task needs to update a given cell, it first has to set this bit to 1 while checking that the bit was not already set to 1 (by another task): the two operations must be performed atomically. In case the bit was already set to 1, we use busy waiting since the cell update is a very fast operation. Another possibility is to suspend the current task and make the underlying thread treat another task: no performance gain was obtained in our tests in doing so. When the update is over, the bit is reset to 0: such write must include a memory barrier to ensure that (i) the write is performed after the computation and that (ii) subsequent reads are performed after this write. With OpenMP, we use `atomic capture` and `atomic update` operations. With TBB, the whole field `val` is declared as an atomic integer and we use a `compare_and_swap` operation to check the bit value and set it to 1. In practice, we expect very few concurrent accesses to the same cells (which may increase the overhead of using atomics operations) since the number of cells in the octree

is many orders of magnitude higher than the number of threads used.

In the `exaFMM-dev` software [13, 15], task parallelism is applied to the DTT differently. In order to avoid conflicts, the traversal is strongly rewritten using one list of children cells for cell A and one list for cell B [13]. These two lists are halved which results in up to four tasks that must be computed among the four half-lists. Task barriers are then used to isolate tasks that can be performed in parallel without conflicts. As mentioned in [13], this implies some extra computations since cells A and B can be opened at the same time, whereas only one cell would have been opened in the original Dehnen's algorithm. Such parallel traversal has also been recently used in [7]. On the contrary, in *pfalcON* we do not require a rewriting of the DTT and we do not introduce extra computations.

Besides, we also have to control the task computation grain for efficient parallel executions. Spawning too small tasks may not enable to offset the task creation overhead, whereas spawning only large tasks may result in an overall load imbalance among threads. We thus introduce a threshold (TCT, *Task Creation Threshold*) to stop task creation: when there are less than TCT particles in the two cells A and B (or in the cell A if $A = B$) no task is created for the remainder of this traversal (but atomic operations are still required). According to the $\mathcal{O}(N)$ runtime complexity, such linear threshold is indeed well-suited to control the computation grain size. Appropriate TCT values for OpenMP and TBB are around 1000 or 10000. It can be noticed that a similar threshold is used in `exaFMM-dev` in order to reduce the number of extra computations introduced by the `exaFMM-dev` task parallelism.

As far as the evaluation step is concerned, the task parallelism is straightforward to implement. We just have to spawn one task at each recursive call to Evaluate (at line 10 in Algorithm 2), since there is no conflict among the tasks. We also use a threshold like TCT in order to control the task computation grain.

3.2 Portable and efficient SIMD direct computation

SPMD model. Many works have already been published on the efficient vectorization of the direct computation for the near field part (see for example [1, 8, 15]). We target here both efficiency and portability on various vector instruction sets (SSE, AVX, Xeon Phi). We thus focus on the SPMD (single-program, multiple-data) model, where all computations are written as scalar ones and it is up to the compiler to merge such scalar computations in SIMD instructions. The main advantages are the ease of programming and the portability: the programmer needs neither to write the specific SIMD intrinsics for each architecture, nor to know the vector width, nor to implement data padding with zeroes according to this vector width. On CPU, such programming model is available in OpenCL (OpenCL implicit vectorization), as well as in the Intel SPMD Program Compiler (`ispc`) [11]. Compared to OpenCL, `ispc` has especially the following advantages [11]: (i) `ispc` kernel launches are faster and (ii) the same memory space and data structures can be shared between the C/C++ code and the `ispc` code. These are very important for *pfalcON* since SIMD computations are performed with small computation grains (usually a few tens of particles per leaf) and require a tight integration in the dual tree traversal and in the octree data structure. We will therefore rely here on the SPMD-on-SIMD model of `ispc`.

`ispc` technical features. C++ classes and templates are not allowed in `ispc` code, which is moreover compiled by a specific compiler. We have thus created in the `ispc` kernels a data structure similar to the *falcON* one for storing particle arrays. The consistency between the two data structures (structure sizes, field offsets, padding) is then checked at runtime. Data alignment is also used like in the original *falcON* code.

In `ispc`, each scalar control flow corresponds to a *program instance* (similar to an OpenCL

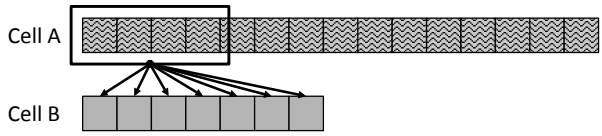


Figure 2: Pair computation: SIMD processing of the first *gs* particles in A (bounding box), successively with each particle in B.

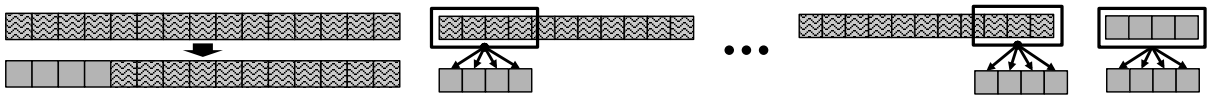


Figure 3: Own computation: pair computation (mutual) between the first particles and the remaining ones, followed by pair computation (no mutual) among the first particles.

work-item). The group of program instances will be merged in one *gang* (similar to a CUDA warp or to an AMD OpenCL wave-front) to be processed concurrently with SIMD instructions. The gang size (denoted *gs*) of the gang is usually set to one or two time(s) the width of the underlying SIMD vector. Depending on the available instruction level parallelism and on the register pressure, it can be indeed more efficient (or not) to use twice the vector width. When the number of items to process is greater than the gang size, the programmer implements the mapping via an explicit loop over all items (contrary to CUDA and OpenCL, where warps/wave-fronts are scheduled by the runtime) [11]. This gives us more control to efficiently and safely implement the direct computation with the mutuality of gravity.

Direct computation kernels. We first focus on the direct computations between two different leafs A and B (*pair* computations). We first determine the leaf with the greatest number of particles (say A here). Each program instance is then in charge of one of the first *gs* particles in A, which leads to a SIMD processing of these *gs* particles (see Fig. 2). Interactions between these *gs* particles and the first particle of B are then processed concurrently: the first particle in B is therefore replicated in the underlying SIMD vector. Force and potential are then updated in A (no conflict among the program instances), as well as in the first particle of B (with `ispc` reductions among the program instances in the gang). This is iterated over all particles in B. Once all particles in B have been treated, the whole process is restarted for the next *gs* particles in A.

We now focus on the direct computations among all particles within one given leaf (*own* computations). In this case, we proceed as in [8] by using as much as possible the (efficient) pair computation along with the mutuality of gravity, and we start by isolating the first *gs* particles. Interactions between these first *gs* particles and the remaining particles are then computed by the pair computation kernel (with the mutuality of gravity): see Fig. 3. After this, the interactions among the first *gs* particles are then computed (similarly to the pair computation, but without the mutuality of gravity here). The whole process is restarted with the remaining particles, whose first *gs* particles are isolated.

Besides, moving from arrays of structures (AoS - as in the scalar *falcON*) to structures of arrays (SoA - more efficient for vector loads and stores) in *falcON* would have required very important programming efforts: we therefore keep the AoS data layout (like in `exaFMM-dev` for example) and rely on the fact that for direct computations the $\mathcal{O}(N)$ memory access times can be rapidly overlapped with the $\mathcal{O}(N^2)$ computation times. Moreover, we also use software pipelining in *pfalcON* with double buffering: we process two interactions at the same time, the first one being computed while data for the next one are being loaded in registers. This has been implemented in both the scalar *pfalcON* (referred to as *pfalcON-scalar*) and the SIMD

	Own		Pair			
	T_1	T_2	$T_{S,1}$	$T_{B,1}$	$T_{S,2}$	$T_{B,2}$
SSE		8	7	3	10	3
AVX	6	72	8	2	32	13
Xeon Phi	3		4	2		

Table 1: SIMD thresholds for each architecture.

code. Finally, we rely on the rsqrt_ps intrinsic SIMD function as a floating-point reciprocal square root estimate, followed by one Newton-Raphson iteration to match floating point single precision.

4 Performance results

For performance tests, we use three compute servers: *SSE-server* with two Intel X5650 CPUs (each having 6 SSE cores with 2-way SMT at 2.67 GHz) and 48 GB of memory, *AVX-server* with two Intel E5-2660 CPUs (each having 8 AVX cores with 2-way SMT at 2.20 GHz) and 32 GB of memory, and *Xeon-Phi* which is a 5110P Xeon Phi (60 cores with 4-way SMT at 1.053 GHz) used in native mode as a distant server. For *pfalcON*, we use OpenMP (3.1) and TBB (4.1) with GCC (4.7.3), since GCC specific optimizations are used in *falcON*, and with ICC (14.0.0) on *Xeon-Phi* for the SIMD intrinsics. `exaFMM-dev`[3] is used with ICC and TBB, and `Bonsai`[4] is run on NVIDIA GPUs (C2070 or K20c, both in *SSE-server*) with CUDA 5.0.

We will use two distributions of 10M particles: an artificial uniform distribution inside a 3D cube, and a Plummer distribution as a classical (non-uniform) astrophysical model [9]. All codes compute both forces and potentials, for all particles, with single precision floating point arithmetic. We also use in each code appropriate softenings for the near field part of the gravity [9].

4.1 SIMD direct computation

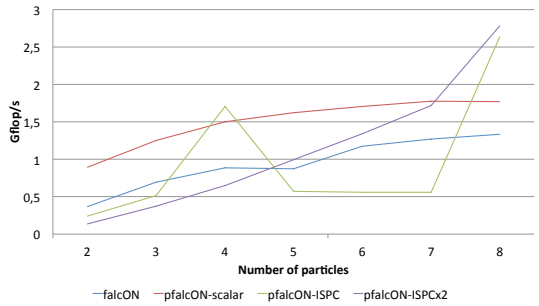
Figure 4 presents the performance of different direct computation kernels presented in Sect. 3.2; namely the original scalar implementation in *falcON*, *pfalcON-scalar* and two SIMD versions in *pfalcON*: with the gang size set to the underlying vector width (*pfalcON-ISPC*) or to twice this vector width (*pfalcON-ISPCx2*). On SSE for example, *pfalcON-ISPC* will rely via `ispc` on a gang size of 4, and *pfalcON-ISPCx2* on a gang size of 8. This last `ispc` feature is however not yet available on the Xeon Phi, where we will therefore only use *pfalcON-ISPC*. The comparison is here performed for numbers of particles that fit in the gang size.

When considering 1 flop for each division and each square root, our scalar code requires 25 flops to compute the forces and potentials between two particles (using the mutuality of gravity). For the sake of the comparison, we use this same number of flops for the SIMD versions (as well as for all subsequent tests).

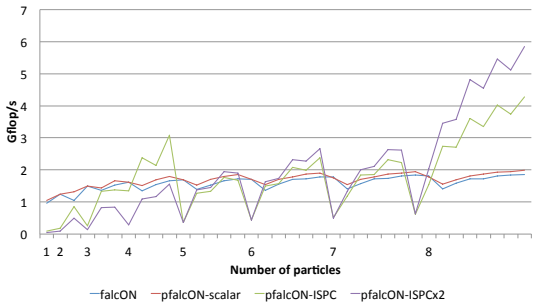
Results in Fig. 4 implies that, depending on the number of particles, especially when there is not enough particles to fill the SIMD vector, it may be better to use our scalar kernel *pfalcON-scalar*, or the SIMD kernel of *pfalcON-ISPC*, instead of the SIMD kernel of *pfalcON-ISPCx2*. We therefore propose, and use hereafter, the following hybrid strategy, based on the underlying vector width (provided by an `ispc` function call).

For own computations of N particles, we first compute the number of particles $N_m = \lfloor N/w \rfloor \times w$ that correspond to multiples of the vector width w, and the remainder $N_r = N - N_m$.

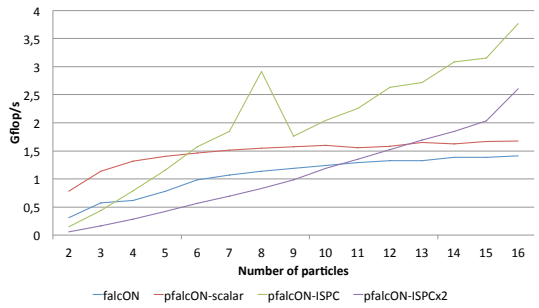
[3] `https://bitbucket.org/rioyokota/exafmm-dev`, commit 4bd77a5, 2013-09-12.
[4] `https://github.com/treecode/Bonsai`, version 8d8e4c0d19, 2013-04-21.



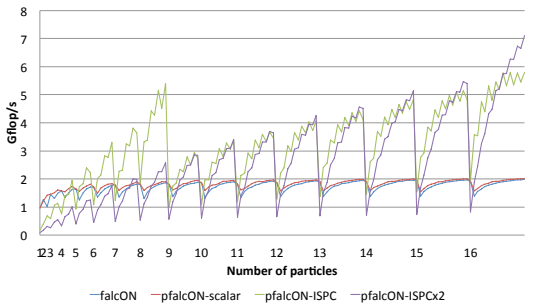
(a) Own computations with SSE



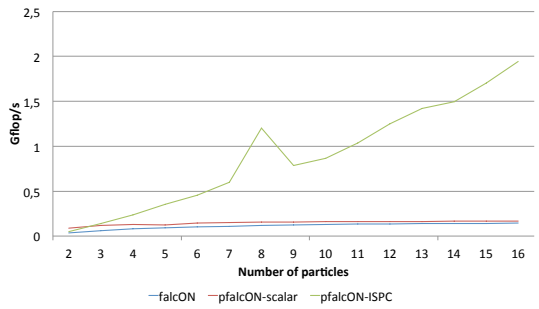
(b) Pair computations with SSE



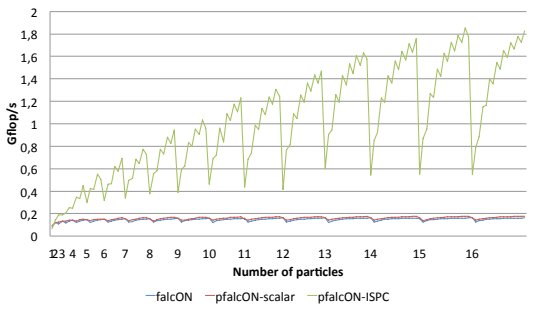
(c) Own computations with AVX



(d) Pair computations with AVX



(e) Own computations on Xeon Phi



(f) Pair computations on Xeon Phi

Figure 4: Performance of the different kernels for direct computations on various SIMD instruction sets. For pair computations between cells A and B, only the number of particles N_A in cell A is indicated: for each value of N_A, N_B ranges from 1 to N_A.

Then we apply specific thresholds for each SIMD architecture to process N_m and N_r. These thresholds are presented in Table 1. For example for N_m, we use:

- *pfalcON-ISPCx2* if $T_2 \leq N_m$;
- *pfalcON-ISPC* if $T_1 \leq N_m < T_2$;
- *pfalcON-scalar* if $N_m < T_1$.

For pair computations of cells A and B with N_A and N_B particles ($N_A \geq N_B$), we first compute $N_{m,A}$ and $N_{r,A}$. We then apply specific thresholds presented in Table 1 following this strategy (firstly for $N_{m,A}$ and secondly for $N_{r,A}$):

- we use *pfalcON-ISPCx2* if $N_{m,A} \geq 2{\times}VectorWidth$ and $N_{m,A} + N_B \geq T_{S,2}$ and $N_B \geq T_{B,2}$;
- otherwise, we use *pfalcON-ISPC* if $N_{m,A} + N_B \geq T_{S,1}$ and $N_B \geq T_{B,1}$;
- otherwise we use *pfalcON-scalar*.

Figures 5(a),5(b),5(c) show that our `ispc` hybrid strategy leads on SSE and AVX to perfor-

mance that is mainly similar or better than the hand-tuned kernels of `exaFMM-dev` [15] for low numbers of particles. For higher numbers of particles, `ispc` clearly outperforms `exaFMM-dev` thanks to a gang size set to twice the vector width. With $N = 1024$ (not shown here), we hence reach for the pair computation 9.5 Gflop/s on SSE and 12.5 Gflop/s on AVX.

On the Xeon Phi, we also obtain similar or better performance than `exaFMM-dev`, except for the high values of N with own computations: this is mainly due to the current lack of *pfalcON-ISPC*x*2* on the Xeon Phi. As soon as a gang size twice larger than the SIMD vector width will be available, we will likely have better performance with `ispc`.

Finally, we evaluate in Fig. 6 the SIMD performance gain on the overall interaction step: values on top of each bar correspond to the speedups of *pfalcON* with `ispc` over the scalar *rfalcON*. We optimally choose here the *Ncrit* value for each code on each architecture: for *rfalcON* the optimal *Ncrit* value is 8, whereas for the SIMD *pfalcON* code this is 32 for AVX and Xeon Phi, and 8 for SSE. The SIMD *pfalcON* code offers thus performance gains over *rfalcON* of 5% on one SSE core, but of up to 24% (resp. 92%) on one AVX (resp. Xeon Phi) core.

4.2 Task parallelism

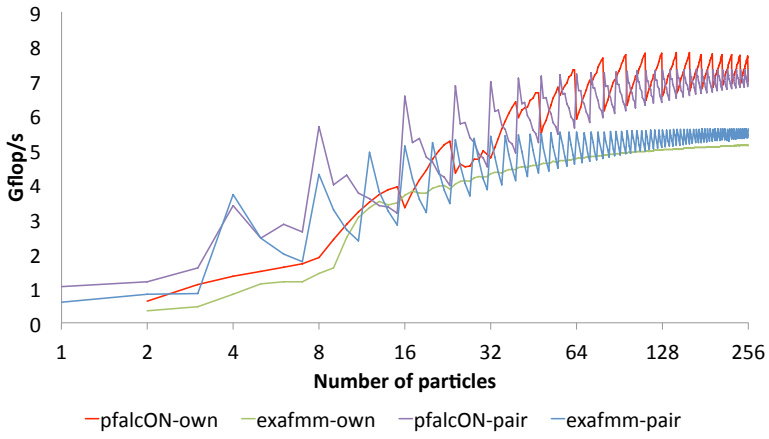
Figure 7 presents speedups of *pfalcON* over the serial *rfalcON* code. With both OpenMP and TBB, and for both uniform and non-uniform distributions of particles, we obtain very good speedups up to 15.8 on *AVX-server* and up to 60 on *Xeon-Phi*. Similar or better parallel efficiencies have been obtained on *SSE-server* (speedups up to 13.8, not shown here). Once the overhead of using task and atomic operations is taken into account (for 1 thread), we indeed obtain linear speedups on up to 32 physical CPU cores. On the Xeon Phi, using two hardware threads per core (denoted as 2-way SMT) enables us to improve the speedup from 50 (with 60 threads) to 60 (with 120 threads on 60 cores). Using two hardware threads is indeed required to reach best performance on this architecture, but performance drops for too many threads (with 4 hardware threads per core - 4-way SMT). These results show that our task parallelism with atomic operations is very well suited for multi-core CPUs, and scales well on the Xeon Phi.

For the evaluation step, good speedups (around 12 on *AVX-server*, and around 32 on *Xeon-Phi*, not shown here) are obtained, these speedups being mainly limited by the very small computation times of this step in our tests.

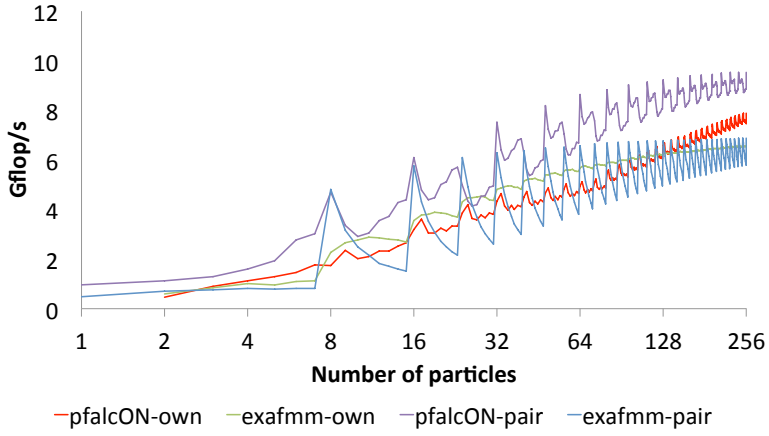
4.3 Comparison with `exaFMM-dev` and `Bonsai`

Finally, we now compare in Fig. 8 the following codes: (i) the original *falcON*, (ii) the SIMD *pfalcON* on *AVX-server* (GCC+OpenMP with 32 hardware threads) and on *Xeon-Phi* (ICC+TBB with 120 hardware threads), (iii) the SIMD `exaFMM-dev` code with 1 or 32 threads on *AVX-server* (ICC+TBB), (iv) and finally `Bonsai` on one C2070 GPU and on one K20c GPU. We compare here two multi-core CPUs (*AVX-server* TDP: $2 \times 95W$) with one GPU (maximum power consumption: 238W for C2070 and 225W for K20c) and with one Xeon Phi (TDP: $225W$), since this corresponds to the same power consumption. Optimal *Ncrit* values are used for *falcON*, *pfalcON* and `exaFMM-dev`, whereas `Bonsai` uses its own specific thresholds ($N_{leaf} = 16$ and $N_{crit} = 64$, see [3]). As recommanded for astrophysical N-body simulations [9], we use $\theta = 0.6$ for *falcON*, *pfalcON* and `exaFMM-dev`, and $\theta = 0.75$ for `Bonsai` (default value) whose expansions and MAC are different [3]. For *falcON*, *pfalcON* and `exaFMM-dev`, we consider here only the interaction and evaluation steps, and for `Bonsai` we consider the corresponding "tree-traverse" step.

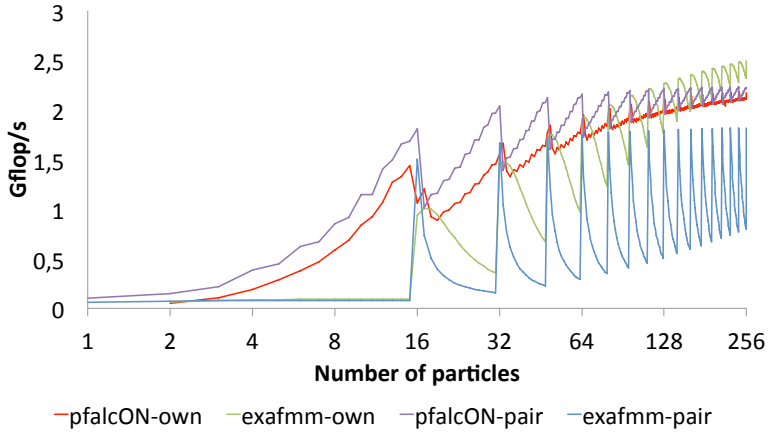
Speedups between *pfalcON* and `exaFMM-dev` are similar. Since *falcON* is somewhat faster for serial executions, *pfalcON* is then also somewhat faster than `exaFMM-dev` for parallel executions.



(a) On one SSE core.



(b) On one AVX core.



(c) On one Xeon Phi core.

Figure 5: Performance comparison for direct computations (only up to 256 particles because of the *Ncrit* limit within each leaf). For pair computations, we use the same number of particles in both cells.

As far as the Xeon Phi is concerned, there is no performance gain compared to the two multi-core CPUs, mainly because astrophysical N-body simulations offer small computation grains

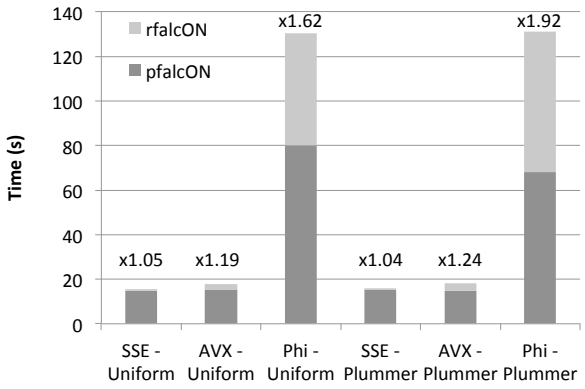
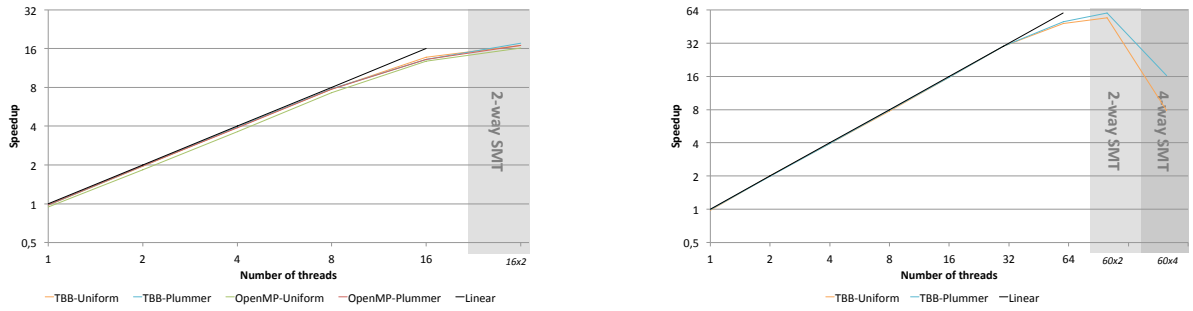


Figure 6: Computation times on one CPU core for the overall interaction step on the two 10M distributions.



(a) On *AVX-server*.

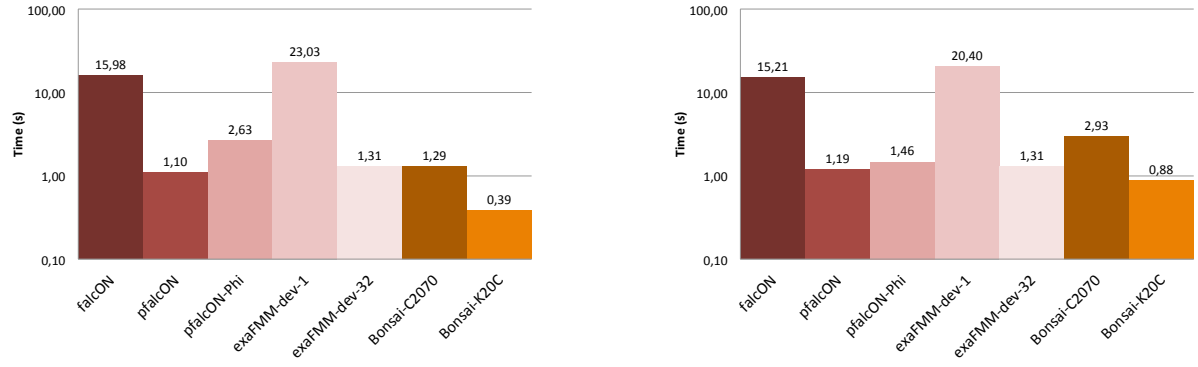
(b) On *Xeon-Phi*.

Figure 7: Speedups obtained by *pfalcON* on multi-core and many-core architectures.

for the direct computations: there is usually too few particles per leaf to fill at best the vector units of the Xeon Phi. The Xeon Phi however outperforms the C2070 for the non-uniform Plummer distribution. Finally, *pfalcON* on *AVX-server* outperforms `Bonsai` on the C2070 GPU. Using the newer K20c GPU, `Bonsai` outperforms *pfalcON* on *AVX-server* for the uniform distribution, but the performance results are much more closer for the more realistic Plummer distribution. Non-uniform distributions are indeed more challenging for GPU codes, whereas *pfalcON* on *AVX-server* is few sensitive to the particle distribution. Lastly, we emphasize that 50M distributions can be run on *SSE-server* and *AVX-server*, but not on any GPU.

5 Conclusion and future work

We have presented a parallel version of the dual tree traversal which is the most challenging and time consuming part in Dehnen's algorithm. Very good speedups are obtained, Dehnen's original algorithm is preserved, no extra computations are introduced, and the SPMD model is shown to be suitable for efficient and portable SIMD vectorization. Since *falcON* is faster than serial Barnes-Hut tree-codes, *pfalcON* with such parallel speedups should outperform any parallel tree-code on one single node with multi-core CPUs. Besides, *pfalcON* is faster than or almost as fast as GPU tree-codes like `Bonsai` for astrophysical distributions, but we emphasize that GPU tree-codes are limited by the GPU memory, and MPI communications on multiple nodes with GPU are usually penalized by the PCI bus. The *pfalcON* code is available at `https://pfalcon.lip6.fr`.



(a) 10M uniform.

(b) 10M Plummer.

Figure 8: Computation times (interaction and evaluation steps) for various N-body codes.

Future work will be focused on the other parts of *falcON* (mainly the octree construction [13]), on distributed-memory parallelism, and on applying such parallel algorithm to other applications than astrophysics. Another (challenging) task would be to efficiently combine the best algorithm, namely the dual tree traversal, with the most powerful hardware currently available, namely GPUs.

References

[1] N. Arora, A. Shringarpure, and R. Vuduc. Direct n-body kernels for multicore platforms. In *Proc. of the Int. Conf. on Parallel Processing (ICPP)*, pages 379–387, 2009.

[2] J. E. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(4):446–449, 1986.

[3] J. Bédorf, E. Gaburov, and S. P. Zwart. A sparse octree gravitational N-body code that runs entirely on the GPU processor. *J. Comp. Phys.*, 231(7):2825–2839, 2012.

[4] M. Burtscher and K. Pingali. An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm. *GPU computing Gems Emerald edition*, page 75, 2011.

[5] H. Cheng, L. Greengard, and V. Rokhlin. A Fast Adaptive Multipole Algorithm in Three Dimensions. *Journal of Computational Physics*, 155:468–498, 1999.

[6] W. Dehnen. A Hierarchical O(N) Force Calculation Algorithm. *J. Comp. Phys.*, 179:27–42, 2002.

[7] W. Dehnen. A fast multipole method for stellar dynamics. 2014. arXiv preprint 1405.2255.

[8] P. Fortin and J.-L. Lamotte. Fast Multipole Method on the Cell B.E.: the Near Field Part. In *Int. Parallel Computing Conf. (ParCo)*, volume 19, pages 323–330, 2009.

[9] Fortin, P., Athanassoula, E., and Lambert, J.-C. Comparisons of different codes for galactic N-body simulations. *Astronomy & Astrophysics*, 531:A120, 2011.

[10] P. Londrillo, C. Nipoti, and L. Ciotti. A parallel implementation of a new fast algorithm for N-body simulations. In *Comp. astro. in Italy: methods and tools*, 2002.

[11] M. Pharr and W. R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13. IEEE, 2012.

[12] V. Springel. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364(4):1105–1134, 2005.

[13] K. Taura, J. Nakashima, R. Yokota, and N. Maruyama. A Task Parallel Implementation of Fast Multipole Methods. In *SC Companion*, pages 617–625, 2012.

[14] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree N-body algorithm. In *Proc. of the 1993 ACM/IEEE conference on Supercomputing*, pages 12–21. ACM, 1993.

[15] R. Yokota. An FMM Based on Dual Tree Traversal for Many-core Architectures. *Journal of Algorithms and Computational Technology*, 7(3):301–324, 2013.