Fine-grained MPI+OpenMP plasma simulations: communication overlap with dependent tasks

Jérôme Richard, Guillaume Latu, Julien Bigot, Thierry Gautier

Fine-grained MPI+OpenMP plasma simulations: communication overlap with dependent tasks

Jérôme Richard[1,2], Guillaume Latu[1], Julien Bigot[3], and Thierry Gautier[4]

[1] CEA/IRFM, F-13108 St-Paul lez Durance, guillaume.latu@cea.fr
[2] Zébrys, Toulouse, jerome.richard@zebrys.fr
[3] Maison de la Simulation, CEA, CNRS, Univ. Paris-Sud, UVSQ, Université Paris-Saclay, julien.bigot@cea.fr
[4] Univ. Lyon, Inria, CNRS, ENS de Lyon, Univ. Claude-Bernard Lyon 1, LIP, thierry.gautier@inrialpes.fr

Abstract. This paper demonstrates how OpenMP 4.5 tasks can be used to efficiently overlap computations and MPI communications based on a case-study conducted on multi-core and many-core architectures. It focuses on task granularity, dependencies and priorities, and also identifies some limitations of OpenMP. Results on 64 Skylake nodes show that while 64% of the wall-clock time is spent in MPI communications, 60% of the cores are busy in computations, which is a good result. Indeed, the chosen dataset is small enough to be a challenging case in terms of overlap and thus useful to assess worst-case scenarios in future simulations.
Two key features were identified: by using task priority we improved the performance by 5.7% (mainly due to an improved overlap), and with recursive tasks we shortened the execution time by 9.7%. We also illustrate the need to have access to tools for task tracing and task visualization. These tools allowed a fine understanding and a performance increase for this task-based OpenMP+MPI code.

Keywords: Dependent tasks · OpenMP 4.5 · MPI · Many-core

1 Introduction

The MPI and OpenMP programming models are widely used in numerical HPC applications [8]. While combining both models is commonplace, several challenges must be addressed to obtain improved performance. One of them is the efficient overlapping of communication with computation since communications are often a major source of overhead. This is critical for future exascale machines expected to interconnect a very large number of computing units.

With the recent shift of HPC platforms from multi-core to many-core architectures, the cumulated communication time can prevail over the computation time [3]. Meanwhile, it can be difficult to keep all cores busy when dealing with fine-grained computations since communication latencies and synchronization costs are possibly an issue at large scale. It is especially challenging as it would

be preferable for parallel applications to provide portable performance on multiple platforms (with reduced development efforts).

Task-based programming is a promising approach to address these problems [16,5]. The introduction of this approach in the version 3.0 of OpenMP has significantly improved the way of expressing parallelism in numerical applications. Especially data dependencies in OpenMP 4.0 (*i.e.* the depend clause), and both task priorities and task loops in the version 4.5 of the norm [11].

As a first contribution, this paper demonstrates that OpenMP tasks can be used to efficiently overlap computations and MPI communications based on a specific case-study on many-core and multi-core architectures. A second contribution is the identification of three specific task parameters that should be carefully tuned to reach this goal: granularity, dependencies and priorities[5]. A third contribution is the proposal of a method based on visualization to understand and guide the tuning of these parameters. Finally, the paper identifies features absent from OpenMP 4.5 that could improve the situation; some of which are already present in OpenMP 5.0 [12].

Section 2 describes the use-case studied in this paper: a hybrid MPI + OpenMP 2D Vlasov-Poisson application while Section 3 discusses its implementation and specifically how we designed algorithms and tasks with communication/computation overlap in mind. Section 4 evaluates and discusses the performance in terms of efficiency, scalability, overlapping; it identifies important task parameters (granularity, dependencies, priorities) and presents a way to adjust these parameters based on tasks visualization. Section 5 discusses related work while Section 6 concludes the paper and presents some future work.

2 Use-Case Description

2.1 Overview and Numerical Approach

Overview We consider an application that solves the Vlasov-Poisson equations[6] to model the dynamics of a system of charged particles under the effects of a self-consistent electric field. The unknown f is a distribution function of particles in the phase space which depends on the time, the physical space, and the velocity. This approach is useful to model kinetically different kinds of plasmas.

Equations The evolution of the distribution function of particles $f(t, x, v)$ in phase space $(x, v) \in \mathbb{R} \times \mathbb{R}$ is given by the Vlasov equation

$$\frac{\partial f}{\partial t} + v \cdot \nabla_x f + E(t, x) \cdot \nabla_v f = 0 \tag{1}$$

In this equation, time is denoted t and the electric field $E(t, x)$ is coupled to the distribution function f. Considering the Vlasov-Poisson system, Poisson is

[5] See [10] for an advanced tutorial about these points.

[6] In practice, Poisson-Ampere[7] are solved instead of Poisson. But for sake of clarity, Poisson-Ampere is not detailed here as the algorithm and performance are very close.

solved in the following way:

$$E(t,x) = -\nabla_x \phi(t,x), \quad -\Delta_x \phi(t,x) = \rho(t,x) - 1\,, \qquad (2)$$
$$\text{with } \rho(t,x) = \int_{\mathbf{R}} f(t,x,v)\,dv$$

where ρ is typically the ionic density, ϕ is the electric potential. One can express the characteristic curves of the Vlasov-Poisson equation (1)-(2) as the solutions of a first-order differential system. It is proven that the distribution function f is constant along the characteristic curves which are the basis of the semi-Lagrangian method we employ to solve the Vlasov equation [17].

Numerical method The semi-Lagrangian method [17] consists in evaluating the distribution function directly on a Cartesian grid in phase space. The driving force of this explicit scheme is to integrate the characteristic curves backward in time at each timestep and to interpolate the value at the feet of the characteristics. The chosen interpolation technique relies on Lagrange polynomials [4] of degree 5. To perform 2D interpolations, we use a tensor product on a fixed-size square region of the 2D grid surrounding the feet of the characteristics.

2.2 Distributed Algorithm and Data Structures

Algorithms Algorithm 1 presents one timestep of the application. It is divided in two parts: the solving of the Poisson equation (lines 1-2) and the Vlasov solver (lines 4-10). The Vlasov solver operates on a 2D regular mesh with ghost areas (data structures are presented in the next paragraph). In the following, we will denote by Δt the time step, by $\rho^n = \rho(n\,\Delta t, x)$ the ionic density at time step n (the superscript notation $.^n$ will be used also for E and f). Ghost exchange is done through `MPI_isend/irecv` between surrounding processes so they can be available for the next iteration. The number of ghost cells in both direction is denoted G. In the upcoming Section 4 we will set $G = 8$.

Algorithm 1: One timestep

Input : f^n, ρ^n_{loc}
Output: f^{n+1}, ρ^{n+1}_{loc}

1 ρ^n = *AllReduce*(ρ^n_{loc})
2 E^n = Field_solver(ρ^n)
3 Perform diagnostics & outputs (E^n, ρ^n)
4 *Launch all isend/irecv for ghost zones* f^n
5 2D advections for interior points (Algo. 2)
6 *Receive wait for ghost zones* f^n
7 2D advections for border points (Algo. 3)
8 *Send wait for ghost zones* f^n
9 ρ^n_{loc} = Local_integral(f^n)
10 Buffer swap between timestep n and $n+1$

Algorithm 2: Interior points advection

Input : Set of local tiles T_* (representing f^n) and E^n
Output: Set of local tiles T_*

1 **for** *k= [indices of local tiles]* **do**
2 **for** $j = [vstart(k) + G : vend(k) - G]$ **do**
3 **for** $i = [xstart(k) + G : xend(k) - G]$ **do**
4 Compute the foot $(x^\star_i, v^\star_j)$ ending at (x_i, v_j);
 `// All needed f values belong to` T_k
5 $f^{n+1}_k(x_i, v_j) \leftarrow$ interpolate $f^n(x^\star_i, v^\star_j)$;

Algorithm 3: Border points advection

Input : Set of local tiles T_* (representing f^n) and E^n
Output: Set of local tiles T_*

1 $D_J = [vstart(k) : vstart(k) + G [\cup] vend(k) - G : vend(k)]$;
2 $D_I = [xstart(k) : xstart(k) + G [\cup] xend(k) - G : xend(k)]$;
3 **for** *k= [indices of local tiles]* **do**
4 **for** $j \in D_J$ **do**
5 **for** $i \in D_I$ **do**
6 Compute the foot $(x^\star_i, v^\star_j)$ ending at (x_i, v_j);
 `// All needed f values belong to` T_k
7 $f^{n+1}_k(x_i, v_j) \leftarrow$ interpolate $f^n(x^\star_i, v^\star_j)$;

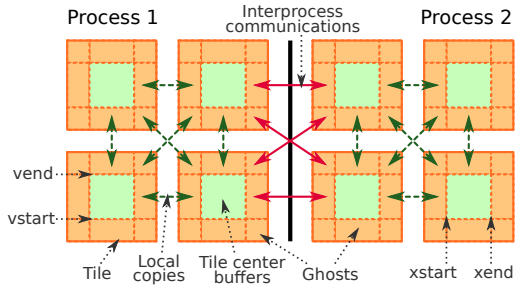


Fig. 1: Tiles and exchange of ghost buffers of the algorithm.

Data Structures The whole (x, v) grid is split into uniform rectangular tiles. Within a tile, two internal buffers are used for double buffering, each buffer corresponding to a timestep. We have also two additional buffers to store ghosts required for the interpolation stencil (virtually surrounding the internal buffers): one for sending data to other tiles and one for receiving data. The ghost area of each tile is split into 8 parts for the 8 neighbor tiles. The tile set is distributed among processes using a 2D decomposition. The points (x, v) of a tile T_k are defined in the domain: $x \in [xstart(k); xend(k)[$ and $v \in [vstart(k); vend(k)[$. The structure of a tile is illustrated in Figure 1 wherein orange areas are ghost zones while light-green areas are internal buffers updated by Vlasov 2D advections.

Computations and Dependencies In Algo. 1, Poisson computation is composed of two sub-steps. First (line l. 1), data is reduced to compute the integral in velocity space (computing ρ as in Eq. 2). Then (l. 2), a local computation is performed in each process. The reduction acts as a synchronization requiring all advection data to be computed before solving Poisson and starting a new step. Thus, most computations of the Vlasov advection and Poisson cannot be overlapped.

Between two advection steps, ghost cells are exchanged with the 8 surrounding tiles (l. 4); data is copied between buffers for tiles that lie in the same process and MPI is used otherwise. The interior points advection (l. 5) can start as soon as Poisson is finished since their interpolation does not depend on ghosts. We assume small displacements (*i.e.* $|v.\Delta t| < \Delta x$ and $|E.\Delta t| < \Delta v$), thus all interpolations can be done locally[7]. Once all ghosts are received for one tile (l. 6), its border points can be computed as well (l. 7).

When the advection of all points of a tile is done, the local part of the integral (needed for the next Poisson computation) is computed (l. 9). Finally, when they are not needed anymore by `MPI_Isend` (l. 8) internal buffers are swapped (l. 10).

Figure 1 displays the communication pattern of the ghost exchange for two MPI processes. Red solid arrows are MPI communications while green dashed arrows are in-process ghost buffers copies. Sent buffers and received buffers are distinct. For sake of clarity, communications/copies providing periodicity along dimensions x and v are not shown here.

3 Implementation Design

We have implemented Algo 1 in C with OpenMP and MPI so as to evaluate the use of OpenMP tasks, priorities and task loops[7] for communication / computation overlap. We have used the OpenMP Tool Interfaces (OMPT) [12] (from OpenMP 5.0) combined to tracing capabilities of the KOMP [5] runtime to analyze the behavior and performance of the code.

Overall design Except where specified otherwise, we use a *flat OpenMP task model* where the *master thread* submits all tasks that are then executed either by *worker threads* or by the master thread itself. This is the only way to create dependencies between sibling tasks [12]. We assign MPI calls and computations to distinct tasks to improve the flexibility of the scheduling and rely on task priorities to guide it. We submit all the tasks of each iteration in batch to provide enough work to feed all workers during a single iteration (critical on many-core systems). The tasks graph is similar on each MPI process.

On the MPI side, we use the `MPI_THREAD_SERIALIZED` mode where all threads can access MPI, but only one at a time. Most MPI implementations use locks to serialize communications in the `MPI_THREAD_MULTIPLE` mode which would interfere with the task model. We instead serialize MPI calls using additional tasks dependencies so that only one MPI task is active at a time. We use non-blocking MPI calls with wait calls in distinct tasks for a fine control of dependencies.

The tasks are submitted with the `omp task` directive and `depend` OpenMP clauses are used to specify the dependencies between them (unless explicitly stated). We decide to prune redundant data dependencies to mitigate submission and scheduling costs (which can be several time higher than the execution if the submission is so slow that workers are starving). For the same reason, we aggregated successive short calculations into single tasks, and consider the whole ghost zone of each tile as a single atomic memory area to achieve coarser granularity.

Algorithm Implementation Poisson is implemented by two tasks. The first one communicates (`MPI_AllReduce`) the density field. The second task depends on the first one and solves Poisson for a local subdomain along space dimension. Once Poisson is solved, a task performs diagnostics: the output of the code.

Ghost buffers management is implemented by two tasks. The first one recursively submits two groups of independent sub-tasks (using synchronous *task loop* construct[8]). Sub-tasks of the first group copy data into ghost buffers and swap tile buffers; those from the second group exchange ghost buffers between local tiles, this avoids MPI transfers that would occur within each MPI process. The second task depends on the first one (and also on the `MPI_AllReduce` task due

[7] See [10] for an advanced tutorial about these points.

[8] This construct specifies to execute iterations of one or multiple loops in parallel using (independent) tasks. Unless specified by the user, it lets the runtime choose the best granularity and perform a final synchronization.

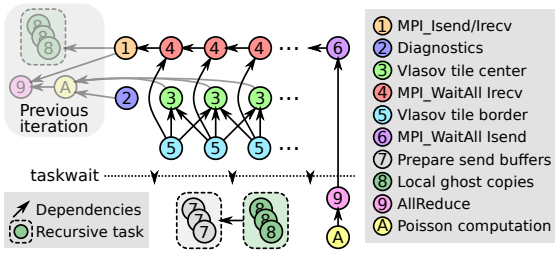


Fig. 2: Submitted task graph of one iteration with the advection (top) and Poisson (bottom) within the implementation.

to MPI tasks serialization) and performs all the `isend`/`irecv` required for tiles whose neighbors are in a different MPI process.

Finally, the advection is implemented by four groups of tasks. Tasks in the first group compute the advected values of internal points of the tiles; each depends only on its own tile both as input and output. Tasks of the second group wait for the reception of ghosts sent in the ghost buffer management tasks. Tasks of the third group compute the advected values of border points and the density integrals ρ; each depends on its own tile both as input and output and on the associated ghost buffer as input. One last task waits for all buffers to be sent. Finally, a `taskwait` directive is performed before moving back to the Poisson phase that depends on the completion of all tasks. MPI tasks have the highest *priority*, then come tile internal point tasks and border point tasks; other tasks have a default lower priority.

Figure 2 summarizes the scheduled OpenMP task graph of one timestep on one MPI process (similar on each process). Firstly, all MPI tasks (top and left) are serialized using an `inout` fake variable. Moreover, while the `MPI_Isend/Irecv` task perform all the Isend/Irecv, each `MPI_WaitAll Irecv` waits only for all ghost buffers of a single tile to be received. For each tile, one task of each of the three following types is submitted: `MPI_WaitAll Irecv`, `Tile center` and `Tile border`. The dependency pattern between the `Tile center` and `Tile border` tasks is a 2D stencil (simplified view on Figure 2).

4 Performance Evaluation

4.1 Experimental setup

System configuration The experiments have been performed on the Skylake (SKL) and Xeon Phi (KNL) partitions of the Marconi supercomputer[9]. SKL nodes include two sockets Xeon 8160 with 24 cores. KNL nodes contain a Xeon Phi KNL 7250 processor with 68 cores. The Xeon Phi is configured with the quadrant clustering and cache memory modes[10]. Hyper-threading is disabled on

[9] `http://www.hpc.cineca.it/hardware/marconi`

[10] The mode cannot be configured by the user on the selected computing machines.

SKL and we chose to disable it on KNL. The network is an Intel Omni-Path 100 Gb/s (fat-tree). Experiments use one process per NUMA node (two processes per socket for SKL and four processes per socket for KNL) to prevent in-process NUMA effects which are out of the scope of this paper (the numactl tool was used so that each process is bound to a unique quadrant and access to its own memory). The code has been compiled with ICC 2018.0.1 and IntelMPI 2018.1.163[11]. The used OpenMP runtime is KOMP [5] (commit 32781b6), a fork of the LLVM/Intel OpenMP runtime. This runtime helps us to produce and visualize runtime traces in order to finely profile and track performance problems and behaviors. It also implements tasks priorities and provides good performance with fine-grained tasks [9].

Method The median completion time is retrieved from a set of 10 runs. Each run performs 1000 iterations for the scaling and granularity experiments and 100 for the trace-based results (to reduce the trace size). Unless explicitly stated, each run works with a (small) tile size of 64x64 over a 2D dataset of 8192x8192. We choose a quite-small dataset for practical reasons: it exhibit issues that usually occur with much more nodes on bigger datasets, but actually takes less time and energy.

4.2 Experimental Results

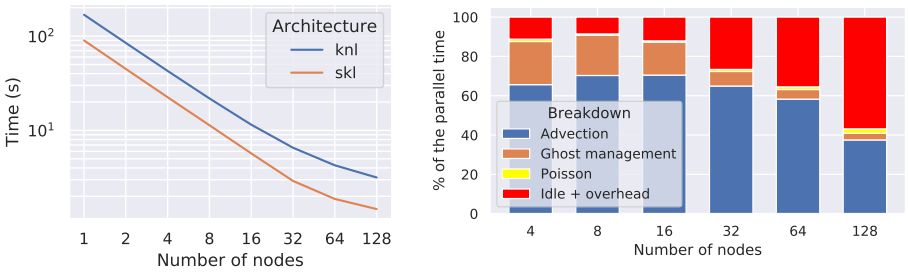
This Section presents and discusses the performance obtained on the considered use-case. First, the overall scalability is analyzed and performance issues are further investigated. Then, the benefits of using priorities is studied. After that, the amount of overlapping is quantified. Finally, task overheads are analyzed: the cost of the submission, the dependencies, and the impact of the task granularity.

Scalability Figure 3a displays the completion time plotted against the number of cores on both KNL and SKL nodes. It shows the hybrid application scales well up to 64 nodes (respectively 4352 and 3072 cores) despite the small amount of data to process. However, some scalability issues appear on 128 nodes nodes (respectively 8704 and 6144 cores).

Breakdown Figure 3b shows the fraction of parallel time (cumulated sum of the duration of the tasks) taken by each part of the application plotted against the number of nodes used. First, we can see that the advection and the ghost exchange needed by the advection take most of the time, while the Poisson part seems negligible at first glance. However, from 32 to 128 nodes, the idle time and the runtime overhead[12] is increasing to the point of becoming dominant, and this growth is mainly due to Poisson solver as we shall see.

[11] The latest available versions on the computing machines during the experiments.

[12] The idle time includes periods where threads are busy waiting for ready tasks to be executed and thread synchronization periods, and the runtime overhead includes scheduling and task submission costs.



(a) Completion time over the number of nodes on each architecture.

(b) Breakdown of the parallel time plotted against the number of SKL nodes.

Fig. 3: Scalability and performance results.

Figure 4 displays the task scheduling of one iteration on one MPI process of 64 SKL nodes (the work is uniformly balanced on each process). Let us focus on Figure 4a for the moment.

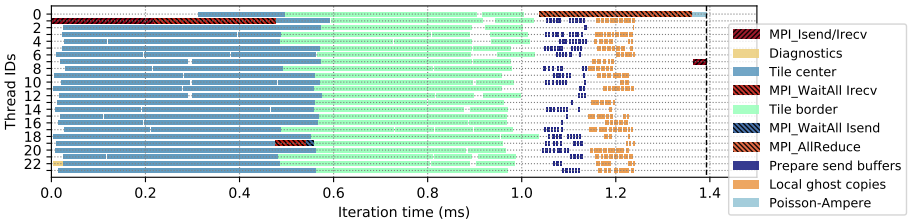
Schedule analysis Overall, the schedule is quite good since tasks related to the advection and ghost exchanges (left side) are feeding almost all cores. The overlap of communication is almost perfect in the Vlasov solver with `MPI_isend/irecv` triggered early in the timestep and the associated `MPI_Wait` do not slow down tiles computations. However, we can see that the Poisson solver (right side) is less effective: the AllReduce does not scale well and the overlapping exists but is dropping along with the number of cores. Indeed, it takes around 30% of the overall execution time on 64 SKL nodes there (and 47% with 128 KNL nodes), with a small fraction overlapped with computation. Please note that this operation only consists in performing communications and can hardly be well overlapped with computations at scale due to the actual dependencies between the steps of Algorithm 1 itself.

It demonstrates the need of overlapping. But, the dependencies of the algorithm prevent any additional overlap with computation in the Poisson solver. Though, the results from all tiles are required to perform the global collective, the result of which is used to solve Poisson, that finally leads to the next timestep.

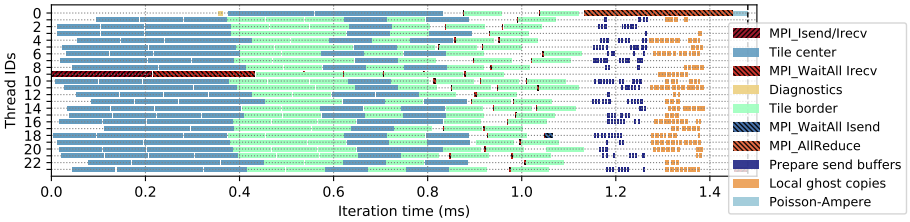
At the beginning of the timestep, we can note that the master thread takes a while before executing tasks. This delay is spent to submit all the tasks for the current timestep and takes a non-negligible part of the time[13].

Task priorities Figure 4b displays the task scheduling with task priorities disabled, as opposed to Figure 4a. This schedule is less efficient since `MPI_Wait` tasks are executed lately reducing the throughput of ready tasks that compute the border points, and at a later stage is delaying the start of Poisson. Please note that the AllReduce is slightly faster. We assume the management of traces

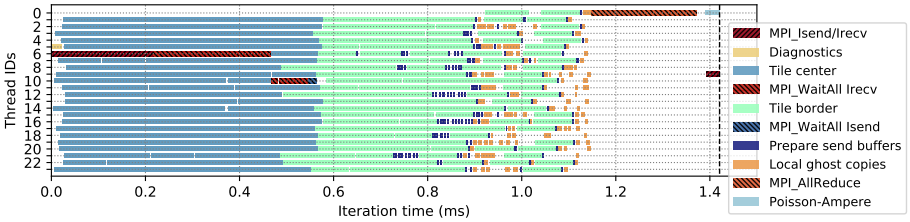
[13] This time could be shortened, if only one could store and resubmit the task graph from one timestep to another such as in [2].



(a) Task scheduling with priorities enabled and task loops.



(b) Task scheduling without priorities enabled and with task loops.



(c) Task scheduling with priorities enabled and without task loops.

Fig. 4: Task scheduling of one iteration within one MPI process using 64 SKL nodes. White areas are idle time and runtime overheads. Dashed lines are the completion time of the selected iterations.

by the runtime causes network transfers that perturbs the AllReduce operation. Overall, this version is 5.7% slower than the version with priorities. Thus, it shows the effectiveness of using task priorities. However, one can note that some OpenMP implementations do not currently support priorities. Indeed, the LLVM/Intel runtime does not yet, but KOMP and GOMP do.

Note that while GOMP could be used in the experiments of this paper, we encountered some limitation when we tried to use it. Indeed, OMPT implemented in GOMP that enable tracing capabilities is still in an early state and we did not succeed to make it work on the tested computing environment (with GCC 8.2.0). Studying results without such information was proven to be tricky.

Quantifying the overlap To measure the amount of overlapping more precisely on the overall execution time, we have designed and used two metrics: r_{comm}

and r_{act}. Let us define $r_{comm} = \frac{t_{comm}}{t_{all}}$ where t_{comm} is the cumulated time of MPI-only tasks (which are serialized) and the t_{all} is the overall completion time. r_{comm} gives hints on whether the application is compute-bound ($r_{comm} \approx 0$) or rather communication-bound ($r_{comm} \approx 1$ with a small r_{act}). $r_{act} = \frac{t_{compute}}{t_{all} \times n_{cores}}$ where $t_{compute}$ is the cumulated sum of all computational task duration (parallel time) and n_{cores} the number of cores. r_{act} represents the amount of activity of all the cores without including communications and runtime overheads.

On Figure 4a, r_{comm} represent the ratio of serialized hatched areas (MPI calls) over the overall completion time and r_{act} is the fraction of non-hatched colored area (OpenMP computing tasks) over the overall two dimensional plotting area (parallel time).

Results show that $r_{comm} = 0.09$ and $r_{act} = 0.96$ on 1 SKL node and $r_{comm} = 0.64$ and $r_{act} = 0.60$ on 64 SKL nodes. It means the application is clearly compute-bound on 1 node as 9% of the overall execution time is spent in MPI calls and cores are busy to perform computation 96% of the time. On 64 nodes, the application spent a major part of its time in MPI calls (64%), but cores are still busy 60% of the time, which indicates a good overlap. This is quite consistent with the schedule of the selected iteration displayed in Figure 4a as serialized MPI calls take a significant portion of the sequential time and the idle time is not predominant although it is clearly significant.

Cost of the dependencies While synchronizations are known to cause load imbalance between threads and could be costly on many-cores systems; the cost of task dependencies can sometimes exceed them. Figure 4c displays the schedule of non-recursive tasks with fine-grained dependencies rather than task loops in recursive tasks of Figure 4a. The task submission takes around 3 times as long as the first version. This large overhead is due to the high number of dependencies (9 per task) compared to the task granularity [9]. Still, the AllReduce is postponed and the master thread is busy at submitting tasks rather than executing them. As a result, the overall completion time of this variant is 9.7% slower than the first one. This justifies the use of task loops in recursive tasks. The overheads are expected to increase with more worker threads or a finer granularity.

More generally, the number of dependencies per task should be minimized[14]. Sometimes, it should be done at the expense of the code maintainability (*e.g.* indirect dependencies, over-synchronizations, control-based dependencies). An analysis of this kind of trade-offs is done in [9].

Impact of the tile granularity Figure 5 displays the overhead coming from the choice of the tile size regarding the number of KNL nodes used. Such overhead includes the management of ghosts, runtime costs (*e.g.* task scheduling), scheduling effect (*e.g.* load balancing), hardware effects (*e.g.* cache effects). A ratio equal to 1 means that the selected tile size provides the best completion time of all evaluated tile sizes for a given number of nodes. A ratio greater than

[14] The management cost of dependencies could also be lowered by the runtime if dedicated studies are done along this line.

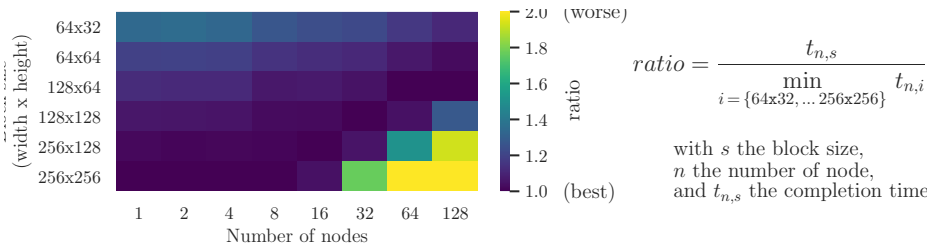


Fig. 5: Relative time (ratio) of the application plotted against both the tile size and the number of nodes of KNL. For each set of nodes, the time is normalized to the best tile size. For clarity, ratios greater than 2 are capped.

1 means that the completion time sub-optimal since another tile size that results in a smaller completion time can be picked.

On one hand, a coarse tile granularity on a lot of nodes (under-decomposition) results in worker starvation since there is not enough tasks to feed them (bottom-right of the Figure). On the other hand, a small tile size on a few nodes (over-decomposition) introduces prohibitive runtime costs due to the number of tasks to schedule proportionally to the overall execution time (top-left of the Figure). Thus, the most efficient tile size is related to the number of nodes.

Strategies to set the tile size We can emphasize that the cost of an under-decomposition of the domain causes more problems than an over-decomposition. Thus, it is better to perform a slight over-decomposition, it provides more flexibility to the runtime to ensure a decent load balance. Finally, choosing 64x64 tiles (see Section 4.1) is not always the best but is an adequate trade-off to ensure the scalability of this code up to 128 nodes.

Discussion on the granularity The tile size, and more generally the granularity of tasks is a matter of concerns to reach good performances. This problem is usually addressed using recursive tasking in OpenMP: tasks can be submitted from different threads and from other tasks recursively to lower the cost of scheduling. However, in OpenMP dependencies can only be defined between sibbling tasks of the same parent task, this is a pitfall to avoid on this use-case. Weak-dependencies proposal [14] would overcome the restriction if well integrated in OpenMP. Since MPI tasks are serialized, they must be submitted from the same thread or parent task and the same rule apply for the advection tasks since they depend on MPI tasks. An alternative is to work at a coarser grain using dependencies on recursive tasks. But, this approach mitigates the submission overhead at the cost of an increased complexity and introduces over-synchronizations that can harm overlapping. Practically, auto-tuning approaches can be used to find the best granularity, especially for more complex use-case.

Leveraging task graph tracing OMPT and the tracing capability of KOMP are the backbone of this paper. First of all, it has guided us to design the application

by providing constant feedback on the runtime scheduling and the source of overheads (dependencies, granularity, synchronizations). For example, it has enabled us to reduce the critical path by tuning the submitted task graph and to improve the overlapping at large scale. Moreover, it also enables the visualization of task scheduling. It proves to be useful to profile the application or even track bugs (*e.g.* bad dependency, abnormal slow task) as well as complex hardware issues (*e.g.* cache effects relative to the locality). This feature also made it possible to draw Figures 4. However, there is no free lunch: tracing introduces an additional overhead which can be significant at fine granularity. That being said, it is still well-suited to analyze the complex behavior of OpenMP task-based applications provided that performance measures are close enough with or without tracing.

Combining tasks and MPI The serialization of MPI communications tasks is more a bypass to prevent issues related to MPI implementations than a definitive solution. Indeed, the developer nor the user are not in the best position to: adjust the number of threads allocated to MPI communications, pin them, deal with issues related to `MPI_THREAD_MULTIPLE`. Indeed, the tuning strongly depends on the MPI implementation, the runtime, the hardware. Moreover, it raises another problem: in which order MPI tasks should be executed to minimize the overall execution time? While we have chosen to force a static schedule of such tasks in the considered use-case, it may not be optimal in general. Indeed, the time of MPI primitives varies regarding the node architectures, the network hardware, the bindings of threads, as well as the actual use of the shared network infrastructure. Moreover, regarding the dependencies and the critical path, it may be worthwhile to start communications before others. OpenMP currently provides no way to deal with such a constrained multi-objective optimization with communication.

5 Related Work

Many studies have been previously conducted on building OpenMP+MPI applications, especially for loop-based applications. But, to our knowledge, no previous work have studied the use of OpenMP 4.5 task-based features on CPU-based many-core systems, especially fine-grained dependencies coupled with MPI.

The MultiProcessor Communications environment (MPC) [13] try to fill in the gap between OpenMP and MPI. While this approach address problems pertaining to the overlapping in hybrid applications, it mainly focuses on loop-based applications. So far, MPC only supports OpenMP 3.1 and thus features like data dependencies, task loops and priorities are not supported. Authors [15] consider to signal blocking MPI calls to OpenMP for better scheduling. The grain of their solution seems to be order of magnitude bigger than considered in our target simulation.

Some task-based runtimes that support OpenMP also support extension relative to MPI. StarPU [1], for example, supports asynchronous and task-based send/receive point-to-point communications, and more recently MPI collectives. The runtime is based on a pooling thread to handle the asynchronous MPI

communications. It provides two main APIs: a C extension based on pragma directives and a low-level C API. The first targets only GCC, and as far as we know, it does not support MPI-related features. The last is flexible, but also more intrusive as a lot of code is needed to submit and manage tasks. OmpSs [6] is another runtime supporting MPI in a similar way. It extends OpenMP with new directives so it can be used by end-users. However, OmpSs relies on its own compiler. These approaches can be a good starting point to create a standard interface between OpenMP and MPI or even an OpenMP extension that could be supported by multiple runtimes as well as MPI implementations.

Although, as of today, designing practical hybrid applications is still challenging for developers adopting both MPI and OpenMP task-based constructs to target recent and upcoming many-core systems. This is an active field of research and the state-of-the-art is moving quickly.

6 Conclusion

This paper evaluated the use of the OpenMP task-related features like priorities and task loops (introduced in the version 4.5 of the norm) in the context of a MPI+OpenMP application that solves the Vlasov-Poisson equations on many-core architectures. It emphasized the impact of using tasks on the overlapping and the overall performance. A specific focus has been put on the tracing and visualization tools. The paper has also highlighted limits specific to OpenMP and provided feedback.

Experiments have been conducted on systems with Skylake and Xeon Phi processors from 1 up to 128 nodes. Results show that OpenMP tasks enable achieving a good overlapping. Task priorities are proven to be effective, especially to schedule MPI communications. The overhead due to tasks submission and due to dependencies management revealed to be quite high. We managed to reduce this overhead by using: less dependencies, task loops constructs and recursive tasks. Finally, OMPT and runtime tracing capabilities have enabled a fine analysis of the behavior and performance of the code, and thus have been essential.

We think that specific points should be mainly addressed in the future. First, the interaction between OpenMP and MPI should be improved in a way the runtime can reorder opportunistically the scheduling of tasks that embed MPI communications. Second, fine-grained dependencies between siblings of recursive tasks should be made possible. Indeed, it is difficult for the user to express all the available parallelism with the existing task features.

Acknowledgments This work was supported by the EoCoE and EoCoE2 projects, grant agreement numbers 676629 & 824158, funded within the EU's H2020 program. We also acknowledge CEA for the support provided by *Programme Transversal de Compétences – Simulation Numérique.*

References

1. Augonnet, C., Aumage, O., Furmento, N., Namyst, R., Thibault, S.: StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. In: Recent Advances in the Message Passing Interface. Springer (2012)
2. Besseron, X., Gautier, T.: Impact of Over-Decomposition on Coordinated Checkpoint/Rollback Protocol. In: et al., A. (ed.) Euro-Par 2011: Parallel Processing Workshops. pp. 322–332. Springer Berlin Heidelberg (2011)
3. Bouzat, N., Rozar, F., Latu, G., Roman, J.: A New Parallelization Scheme for the Hermite Interpolation Based Gyroaverage Operator. In: 2017 16th ISPDC (2017)
4. Bouzat, N., et al.: Targeting Realistic Geometry in Tokamak Code Gysela. ESAIM: Proceedings and Surveys **63**, 179–207 (2018)
5. Broquedis, F., Gautier, T., Danjean, V.: LIBKOMP, an Efficient OpenMP Runtime System for Both Fork-join and Data Flow Paradigms. In: Proceedings of IWOMP 2012. Springer (2012)
6. Bueno, J., et al.: Productive Cluster Programming with OmpSs. In: European Conference on Parallel Processing. Springer Berlin Heidelberg (2011)
7. Crouseilles, N., Latu, G., Sonnendrücker, E.: Hermite spline interpolation on patches for parallelly solving the Vlasov-Poisson equation. IJAMCS **17**(3) (2007)
8. Diaz, J., Muñoz-Caro, C., Niño, A.: A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. IEEE TPDS **23**(8), 1369–1386 (2012)
9. Gautier, T., Pérez, C., Richard, J.: On the Impact of OpenMP Task Granularity. In: Proceedings of IWOMP 2018. pp. 205–221. Springer (2018)
10. Martorell, Xavier and Teruel, Xavier and Klemm, Michael: Advanced OpenMP Tutorial (2018), `https://openmpcon.org/wp-content/uploads/2018_Tutorial3_Martorell_Teruel_Klemm.pdf`
11. OpenMP Architecture Review Board: OpenMP Application Programming Interface Version 4.5 (Nov 2015), `http://www.openmp.org`
12. OpenMP Architecture Review Board: OpenMP Application Programming Interface Version 5.0 (Nov 2018), `http://www.openmp.org`
13. Pérache, M., Jourdren, H., Namyst, R.: MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In: Euro-Par 2008 Parallel Processing. Springer (2008)
14. Perez, J.M., Beltran, V., Labarta, J., Ayguadé, E.: Improving the Integration of Task Nesting and Dependencies in OpenMP. In: IPDPS 2017. IEEE (2017)
15. Sala, K., et al.: Improving the Interoperability Between MPI and Task-Based Programming Models. In: Proceedings of EuroMPI 2018. pp. 6:1–6:11. ACM (2018)
16. Song, F., YarKhan, A., Dongarra, J.: Dynamic Task Scheduling for Linear Algebra Algorithms on Distributed-memory Multicore Systems. In: Proceedings of the Conference on HPC Networking, Storage and Analysis. SC'09, ACM (2009)
17. Sonnendrücker, E., et al.: The semi-Lagrangian method for the numerical resolution of the Vlasov equation. Journal of Computational Physics **149**(2), 201 – 220 (1999)