InKS, a Programming Model to Decouple Performance from Algorithm in HPC Codes

Ksander Ejjaaouani[1,2(✉)], Olivier Aumage[3,4], Julien Bigot[1], Michel Mehrenberger[5], Hitoshi Murai[6], Masahiro Nakao[6], and Mitsuhisa Sato[6]

[1] Maison de la Simulation, CEA, CNRS, Univ. Paris-Sud, UVSQ Université Paris-Saclay, Gif-sur-Yvette, France
julien.bigot@cea.fr
[2] Inria, Nancy, France
ksander.ejjaaouani@inria.fr
[3] Inria, Bordeaux, France
olivier.aumage@inria.fr
[4] LaBRI, Bordeaux, France
[5] IRMA, Université de Strasbourg, Strasbourg, France
mehrenbe@math.unistra.fr
[6] Riken AICS, Kobe, Japan
{h-murai,masahiro.nakao,msato}@riken.jp

Abstract. Existing programming models tend to tightly interleave algorithm and optimization in HPC simulation codes. This requires scientists to become experts in both the simulated domain and the optimization process and makes the code difficult to maintain and port to new architectures. This paper proposes the InKS programming model that decouples these two concerns with distinct languages for each. The simulation algorithm is expressed in the InKS_{pia} language with no concern for machine-specific optimizations. Optimizations are expressed using both a family of dedicated optimizations DSLs (InKS_{O}) and plain C++. InKS_{O} relies on the InKS_{pia} source to assist developers with common optimizations while C++ is used for less common ones. Our evaluation demonstrates the soundness of the approach by using it on synthetic benchmarks and the Vlasov-Poisson equation. It shows that InKS offers separation of concerns at no performance cost.

Keywords: Programming model · Separation of concerns
HPC · DSL

1 Introduction

It is more and more common to identify simulation as the *third pillar of science* together with theory and experimentation. Parallel computers provide the computing power required by the more demanding of these simulations. The complexity and heterogeneity of these architectures do however force scientists to write complex code (using vectorization, parallelization, accelerator specific languages, etc.) These optimizations heavily depend on the target machine and the whole code has to be adapted whenever it is ported to a new architecture.

As a result, scientists have to become experts in the art of computer optimizations in addition to their own domain of expertise. It is very difficult in practice to maintain a code targeting multiple distinct architectures. One fundamental cause for this situation is the tight interleaving of two distinct concerns imposed by most programming models. On the one hand, the algorithm comes from the expertise of the domain scientist and does not depend on the target architecture. On the other hand, optimization is the expertise of optimization specialists and has to be adapted for every new architecture.

Many approaches have been proposed to improve this situation in the form of libraries or languages. Approaches based on automated optimization processes typically isolate the algorithmic aspects very well but restrict their domain of applicability and/or the range of supported optimizations. Approaches based on optimization tools and libraries enable optimization specialists to express common optimizations very efficiently but leave others mixed with the algorithm.

In this paper, we propose the Independent Kernel Scheduling (InKS) programming model to separate algorithm from optimization choices in HPC simulation codes. We define the InKS_{pia} language used to express the algorithm of an application independently of its optimization. Such a program can be optimized with the InKS_{O} family of domain-specific languages (DSLs) for common optimizations or C++ for less common optimizations.

This paper makes the following contributions: **(1)** it defines the InKS programming model and its *platform-independent algorithmic* language InKS_{pia}; **(2)** it proposes an implementation of InKS with two optimization DSLs, $\text{InKS}_{\text{O/Loop}}$ and $\text{InKS}_{\text{O/XMP}}$; and **(3)** it evaluates the approach on the synthetic NAS parallel benchmarks [3] and on the 6D Vlasov-Poisson solving with a semi-Lagrangian method.

The remaining of the paper is organized as follows. Section 2 presents and analyzes related work. Section 3 describes the InKS programming model and its implementation. Section 4 presents the 6D Vlasov-Poisson problem and its implementation using InKS while Sect. 5 evaluates the approach. Finally, Sect. 6 concludes the paper and identifies some perspectives.

2 Related Works

Many approaches are used to implement optimized scientific simulations. A first widely used approach is based on imperative languages such as Fortran, C or C++. Libraries like MPI extend this to distributed memory with message passing. Abstractions very close to the actual execution machine make fine-tuning possible to achieve good performance on any specific architecture. It does however require encoding complex optimizations directly in the code. As there is no language support to separate the algorithm and architecture-specific optimizations, tedious efforts have to be applied [10] to support performance portability. Algorithm and optimizations are instead often tightly bound together in codes.

A second approach is thus offered by tools (libraries, frameworks or language extensions) that encode classical optimizations. *OpenMP* [5] or *Kokkos* [4] support common shared-memory parallelization patterns. *E.g*, Kokkos offers multi-dimensional arrays and iterators for which efficient memory mappings and iteration orders are selected independently. *E.g*, *UPC* [8] or XMP [14] support the partitioned global address space paradigm. In XMP, directives describe array distribution and communications between nodes. These tools offer gains of productivity when the optimization patterns they offer fit the requirements. The separation of optimizations from the main code base also eases porting between architectures. Even if expressed more compactly optimizations do however remain mixed with the algorithm and only cover part of the requirements.

A third approach pushes this further with tools that automate the optimization process. For example, *PaRSEC* [11] or *StarPU* [1] support the many-tasks paradigm. In StarPU, the user expresses its code as a DAG of tasks with data dependencies that is automatically scheduled at runtime depending on the available resources. Another examples are *SkeTo* [18] or *Lift* [16] that offer algorithmic skeletons. Lift offers a limited set of parallel patterns whose combinations are automatically transformed by an optimizing compiler. Automating optimization improves productivity and clearly separate these optimizations which improves portability. The tools do however not cover the whole range of potential optimizations such as the choice of work granularity inside tasks. The algorithm remains largely interleaved with optimization choices even with this approach.

A last approach is based on DSLs that restrict optimizations, such as *Pochoir* [17] or *PATUS* [6], DSLs for stencil problems. In Pochoir, the user only specifies a stencil (computation kernel and access pattern), boundary conditions and a space-time domain while all optimizations are handled by a compiler. DSLs restrict the developer to the expression of the algorithm only, while optimizations are handled independently. This ensures a very good separation of these aspects. The narrower the target domain is, the more efficient domain and architecture-specific optimizations are possible. However, it makes it less likely for the tool to cover the needs of a whole application. Real-world applications can fall at the frontier between the domains of distinct DSLs or not be covered by a single one. Performance then depends on the choice of DSLs to use and the best choice depends on the target architecture leading to new portability issues.

To summarize, one can consider a continuum of approaches from very general approaches where the optimization process is manual to more and more domain specific where the optimization process can be automated. The more general approaches support a large range of optimizations and application domains but yield high implementation costs and low separation of concerns and portability. The more automated approaches reduce implementation costs and offer good separation of concerns and portability but restrain the range of supported domains and optimizations. Ideally, one would like to combine all these advantages: **(1)** the domain generality of imperative languages, **(2)** the ease of optimization offered by dedicated tools and **(3)** the separation of concerns and performance portability offered by DSLs. The following section describes the InKS programming model that aims to combine these approaches to offer such a solution.

3 The InKS Programming Model

This section describes the core of our contribution, the InKS programming model. This approach is based on the use of distinct languages to express algorithm and optimization choices; thus enforcing their separation. The *algorithm* of the simulation consists in the set of values computed, the formula used to produce each of them as well as the simulation inputs and outputs. We include in *optimization choices* all that is not the algorithm, such as the choice of a computing unit for each computation, their ordering, the choice of a location in memory for each value, etc. Multiple optimization choices can differ in performance but simulation results depend on the algorithm only.

The InKS approach is summarized in Fig. 1. The InKS$_{\mathsf{pia}}$ language is used to express the algorithm with no concern for optimization choices. A compiler generates non-optimized, generic choices automatically from this specification for test purposes. The InKS$_{\mathsf{O}}$ family of DSLs is used to define common optimizations efficiently while C++ is used to describe arbitrarily complex optimizations. Many versions of the optimization choices can be devised to optimize for multiple targets.

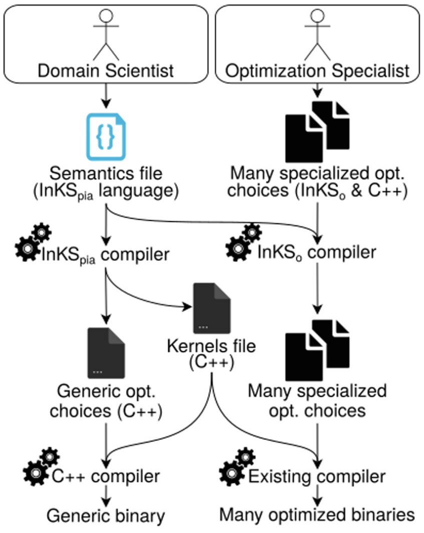


Fig. 1. The InKS model

The remaining of the section describes InKS$_{\mathsf{pia}}$ and proposes two preliminary InKS$_{\mathsf{O}}$ DSLs. InKS$_{\mathsf{O/XMP}}$ handles domain decomposition and inter-node communications while InKS$_{\mathsf{O/Loop}}$ focuses on efficient loops.

The InKS$_{\mathsf{pia}}$ *language.* In InKS$_{\mathsf{pia}}$ [2] (illustrated in Listing 1), values are stored in infinite multidimensional arrays based on dynamic single assignment (DSA, each coordinate can only be written once). Memory placement of each coordinate is left unspecified. Computations are specified by *kernel* procedures that **(1)** take as parameters data arrays and integer coordinates; **(2)** specify the coordinate they might read and will write in each array; and **(3)** define either a C++ or InKS implementation. An InKS implementation defines kernels *validity domains*: coordinates where C++ kernels can generate values in arrays. Kernel execution order is left unspecified. The simulation entry point is a kernel marked *public*. This specifies a parameterized task graph (PTG) [7]. This representation covers a large range of problems but imposes a few limitations. Mostly, all the problem parameters must be known at launch time, it does for example not support adaptive mesh or time-steps. It is still possible to express these concerns outside InKS$_{\mathsf{pia}}$ and call the InKS implementation multiple times with different parameters.

The InKS$_{\mathsf{pia}}$ compiler [2] extracts C++ functions from InKS$_{\mathsf{pia}}$ kernels and generates a function with correct but non-optimized loops and memory allocations to execute them. Arbitrarily complex versions of these can also be written manually in plain C++ and rely on existing optimization tools. These functions can be called from any existing code whose language supports the C calling convention. However, that approach requires information present in InKS$_{\mathsf{pia}}$ to be repeated. The InKS$_{\mathsf{O}}$ DSLs thus interface optimization tools with InKS$_{\mathsf{pia}}$.

```
kernel stencil3(x, t): ( double A(2) {in: (x-1:x+1, t) | out: (x, t+1)} )
    #CODE(C) A(x, t+1) = 0.5*A(x, t-1) + 0.25*(A(x-1, t-1)+A(x+1, t-1)); #END
public kernel inks_stencil(DIM_X, N_ITER):
        ( double A_decl(2) {in: (0:DIM_X, 0) | out: (1:DIM_X-1, N_ITER-1)} )
    #CODE(INKS) stencil3(1:DIM_X-1, 1:N_ITER-1):(A_decl) #END
```

Listing 1: 1D stencil computation on a 2D domain in InKS$_{\mathsf{pia}}$

```
#pragma xmp nodes p6d[pV3][pV2][pV1][pZ][pY][pX]
#pragma inks decompose dynamic % f6d // dynamic allocation of f6d algorithmic array
    (8:, 7:, 6, 5, 4, 3, 2, 1) // dimension reordering, dim 7 & 8 are folded
    with t6d onto p6d // block decomposition mapped on the XMP topology
// Dynamic halo exchange on the 4th dimension, halo sizes are computed automatically
#pragma inks exchange periodic f6d(4, advection4) to R and L
/*R and L are now allocated buffers and contain the halo values: */ foo(R, L);
```

Listing 2: 6D decomposition and dynamic halo exchange in InKS$_{\mathsf{O/XMP}}$

```
double (f6d*)[][][][][]; // need to declare f6d global, valid in xmp
#pragma xmp nodes p6d[pV3][pV2][pV1][pZ][pY][pX] // xmp 6d cartesian node topology
#pragma xmp template t6d[:][:][:][:][:][:] // xmp 6d logical array
#pragma xmp distribute t6d [block][block][block][block][block][block] onto p6d
// map element of f6d to element of t6d
#pragma xmp align f6d[n][m][l][k][j][i] with t6d[n][m][l][k][j][i]
#pragma xmp template_fix t6d[N][M][L][K][J][I]
f6d = (double(*)[M][L][K][J][I]) xmp_malloc(xmp_desc_of(f6d), N, M, L, K, J, I);
```

Listing 3: XMP code generated for the `f6D` decomposition of Listing 2

The InKS$_{\mathsf{O/XMP}}$ *Optimization Language.* InKS$_{\mathsf{O/XMP}}$ (illustrated in Listing 2) handles distributed memory domain decomposition by combining C and directives based on XMP and adapted for InKS. The compiler replaces these directives by C and XMP code (Listing 3). The `inks decompose` directive supports static or dynamic allocation of arrays from the algorithm. The domain size is extracted from InKS$_{\mathsf{pia}}$ source and the user only has to specify its mapping onto memory. As in XMP, InKS$_{\mathsf{O/XMP}}$ supports domain decomposition mapped onto an XMP topology. In addition, it supports dimension reordering and folding which consists in reusing the same memory address for subsequent indices in a given dimension. This feature is important to reuse memory due to the DSA nature of InKS$_{\mathsf{pia}}$ arrays. The `exchange` directive supports halo exchanges. The required halo size is automatically extracted from the algorithm and the user only has to specify when to execute the communications and in what dimension. While XMP requires halo values to be stored contiguously with the domain, InKS$_{\mathsf{O/XMP}}$ support a dynamic halo extension where halo values are stored in dedicated, dynamically allocated buffers to reduce memory footprint.

```
/*** advec.iks InKSpia algorithmic file ***/
kernel advection3 (i, j, k, l, m, n, t, K, step) : (
   double f6d { in: (i, j, 0:K, l, m, n, t, step-1) | out: (i, j, k, l, m, n, t, step) },
   int disp {in: (n)}, double coef {in: (n, 0:4)} )
#CODE (C) /* ... */ #END

public kernel main_code(t, I, J, K, L, M, N, Niter): ( double coef(2), int disp(1)
   double f6d(6) { in: (i, j, 0:K, l, m, n, t, step-1) | out: (i, j, k, l, m, n, t,
        step) } )
#CODE(INKS)
  /* ... */
  advection3 (0:I, 0:J, 0:K, 0:L, 0:M, 0:N, 1:Niter, K, 2) : (f6d, disp, coef)
  /* ... */
#END

/*** advec.iloop InKSo/Loop optimization choices file ***/
loop advection3_loops(t, I, J, K, L, M, N, Niter) : advection3 { // set "t" value
  // "Set" not specified -> loop bounds are computed, with a fixed "t"
  Order: n, m, l, j, i, k; // order of the loop
  Block: 16; // blocking on the inner dimension k
  Buffer: f6d(3); } // copy the third dimension of f6d to a 1d buffer
```

Listing 4: A loop nest in InKS$_{\mathsf{pia}}$ optimized in InKS$_{\mathsf{O/Loop}}$

```
/* for all N, M, L, J, do */ for (int i=0; i<I; i+=blockSize){
  for(int ci=-halo_size; ci<0; ++ci) /*Copy to buffer*/
    for(int ii=0; ii<blockSize; ++ii) buff.buff_in[...] = left[...];
  for(int ci=0; ci<K; ++ci)
    for(int ii=0; ii<blockSize; ++ii) buff.buff_in[...] = f6d[...];
  for(int ci=sizeK; ci<sizeK+halo_size; ++ci)
    for(int ii=0; ii<blockSize; ++ii) buff.buff_in[...] = right[...];
  for(int idb=0; idb<size_block; ++idb) /*Computation*/
    for(int k=0; k<K; ++k) advection3(buff, idb+i, j, k, l, m, n);
  for(int ci=0; ci<K; ++ci) /*Copy to f6d*/
    for(int ii=0; ii<block_size; ++ii) f6d[...] = buff.buff_out[...];}
```

Listing 5: C++ code generated for the loop nest of Listing 4

The InKS$_{\mathsf{O/Loop}}$ *Optimization Language.* InKS$_{\mathsf{O/Loop}}$ (illustrated in Listing 4) offers to specify manually loop nests for which the compiler generates plain C++ loops (Listing 5). Plain C++ is usable with InKS$_{\mathsf{O/Loop}}$. The `loop` keyword introduces a nest optimization with a name, the list of parameters from the algorithm on which the loop bounds depend and a reference to the optimized kernel. Loop bounds can be automatically extracted from InKS$_{\mathsf{pia}}$, but the `Set` keyword makes it possible to restrict these bounds. The `Order` keyword specifies the iteration order on the dimensions and the `Block` keyword enables the user to implement blocking. It takes as parameters the size of block for the loops starting from the innermost one. If there are less block sizes than loops, the remaining loops are not blocked. The `Buffer` keyword supports copying data in a local buffer before computation and back after to ensure data continuity and improve vectorization. The compiler uses data dependencies from the algorithm to check the validity of the loops order and generate vectorization directives where possible.

4 The 6D Vlasov/Poisson Problem

The 6D Vlasov-Poisson equation, presented in (1), describes the movement of particles in a plasma and the resulting electric field. We study its resolution for a single species on a 6D Cartesian mesh with periodic boundary conditions. We solve the Poisson part using a fast Fourier transform (FFT) and rely on a Strang splitting (order 2 in time) for the Vlasov part. This leads to 6 1D advections: 3 in the space dimensions (x_1, x_2, x_3) and 3 in the velocity dimensions (v_1, v_2, v_3). Each 1D advection relies on a Lagrange interpolation of degree 4. In the space dimensions, we use a semi-Lagrangian approach where the stencil is not applied around the destination point but at the foot of characteristics, a coordinate known at runtime only. This is described in more details in [15].

$$\begin{cases} \dfrac{\partial f(t,x,v)}{\partial t} + v.\nabla_x f(t,x,v) - E(t,x).\nabla_v f(t,x,v) = 0 \\ -\varDelta\phi(t,x) = 1 - \rho(t,x) \\ E(t,x) = -\nabla\phi(t,x) \\ \rho(t,x) = \displaystyle\int f(t,x,v)dv \end{cases} \tag{1}$$

The main unknown is f (`f6D` in the code), the distribution function of particles in 6D phase space. Due to the Strang splitting, a first half time-step of advections is required after `f6D` initialization but before the main time-loop. These advections need the electric field E as input. E is obtained through the FFT-based Poisson solver that in turn needs the charge density ρ as input. ρ is computed by a reduction of `f6D`. The main time-loop is composed of 3 steps: advections in space dimensions, computation of the charge density (reduction) and electric field (Poisson solver) and advections in velocity dimensions.

The 6D nature of `f6D` requires a lot of memory, but the regularity of the problem means it can be distributed in blocks with good load-balancing. Halos are required to hold values of neighbors for the advections. Connected halo zones would increase the number of points in all dimensions and consume too much memory. Split advections mean that halos are required in a single dimension at a time though. We therefore use dynamic halos composed of two buffers, one for each boundary of the advected dimension (denoted *"right"* and *"left"*) as shown on Fig. 2. Listing 2 corresponds to the InKS$_{\mathsf{O/XMP}}$ implementation of this strategy.

Advections are the main computational cost of the problem, accounting for 90% of the sequential execution time. Six loops surround the stencil computation of each advection and in a naive version, the use of a modulo to handle periodicity and application along non-contiguous dimensions slow down the computation. To enable vectorization and improve cache use, we copy `f6D` elements into contiguous buffers along with the *left* and *right* halos. Advections are applied on these buffers before copying them back into `f6D`. Blocking further improves performance by copying 16 elements at a time. Listing 5 corresponds to the InKS$_{\mathsf{O/Loop}}$ implementation of these optimizations.

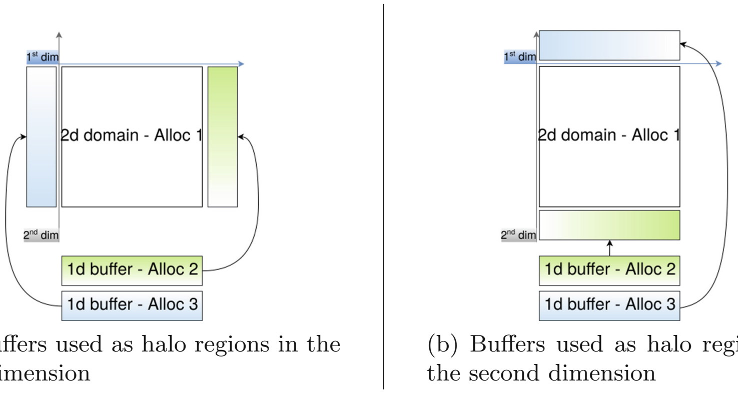


Fig. 2. Dynamic halo exchange representation on a 2D domain

5 Evaluation

This section evaluates the InKS model on the NAS parallel benchmark, a simple stencil code and the 6D Vlasov-Poisson problem. Plain C++ is used for the synthetic benchmarks optimization while InKS$_{\mathsf{O/XMP}}$, InKS$_{\mathsf{O/Loop}}$, plain C++ and C with XMP are used for Vlasov-Poisson. All codes are compiled with Intel 18 compiler (`-O3 -xHost`), Intel MPI 5.0.1 and executed on the *Poincare* cluster (Idris, France) with 32 GB RAM and two Sandy Bridge E5-2670 CPUs per node and a QLogic QDR InfiniBand interconnect.

5.1 Synthetic Benchmarks

We have implemented 4 out of the 5 sequential NAS benchmark kernels (`IS`, `FT`, `EP` and `MG`) in InKS$_{\mathsf{pia}}$ and optimized them with plain C++. The C++ version [9], is used as reference. We have also implemented a 3D heat equation resolution by finite differences (7-point stencil) with two distinct C++ optimization strategies from a single InKS$_{\mathsf{pia}}$ source. Both strategies comes from the reference [12]. One uses double buffering (Heat/Buf) and the other implements a cache oblivious strategy (Heat/Obl).

The NAS `CG` kernel relies on indirections not expressible in the PTG model of InKS$_{\mathsf{pia}}$. Its implementation would thus have to rely on a large C++ kernel whose optimization would be mixed with the algorithm. InKS$_{\mathsf{pia}}$ can however be used to express all other NAS kernels as well as the 3D heat equation solver. Even if not as expressive as C or Fortran, InKS$_{\mathsf{pia}}$ covers the needs of a wide range of simulation domains and offers abstractions close to the execution machine rather than from a specific simulation domain. Among others, it can express computations such as FFTs or stencils with input coordinates unknown at compile-time.

InKS separates the specification of algorithm and optimization in distinct files. Multiple optimization strategies can be implemented for a single algorithm,

Table 1. Execution time of the C++ and InKS implementations of the sequential NAS benchmark, class B - time/iteration of the 3D heat equation (7point stencil), size (1024^3) - median and standard deviation of 10 executions - GNU complexity score of the implementation.

	Execution time (second)			Complexity	
Benchmark	Reference	InKS	Rel. dev.	Ref.	InKS
NAS/FT	62.43 (±0.57)	62.86 (±0.71)	0.68%	6	5
NAS/IS	3.39 (±0.00)	3.44 (±0.00)	1.47%	55	52
NAS/MG	5.10 (±0.02)	4.73 (±0.06)	−7.25%	20	12
NAS/EP	76.43 (±0.21)	76.47 (±0.22)	0.05%	19	19
Heat/Buf	3.05 (±0.01)	2.97 (±0.06)	−2.58%	5	3
Heat/Obl	2.43 (±0.01)	2.05 (±0.02)	−15.59%	22	13

as shown for the 3D heat equation where each relies on a specific memory layout and scheduling. It thus offers a clear separation of algorithm and optimization.

Finding the right metric to evaluate the easiness of writing a code is a difficult question. As illustrated in Listing 4 however, algorithm expression in InKS$_{\text{pia}}$ is close to the most naive C implementation where loops are replaced by InKS validity domains with no worry for optimization. The specification of optimization choices is close to their expression in C++. Table 1 compares the GNU complexity score of InKS optimizations to the reference code. InKS scores are slightly better because kernels extracted from the algorithm hide computations and thus, part of the complexity. In addition, the use of C++ to write optimizations let optimization specialists reuse their preexisting knowledge of this language. These considerations should not hide the fact that some information has to be specified both in the InKS$_{\text{pia}}$ and C++ files with this approach leading to more code overall.

Regarding performance, the InKS approach makes it possible to express optimizations that do not change the algorithm. Optimizations of the four NAS parallel benchmarks and 3D heat equation solver in InKS were trivial to implement and their performance match or improve upon the reference as presented in Table 1. Investigation have shown that Intel ICC 18 does not vectorize properly the reference versions of Heat/Obl and NAS/MG. The use of the Intel `ivdep` directive as done on the InKS versions leads to slightly better performance.

5.2 Vlasov-Poisson

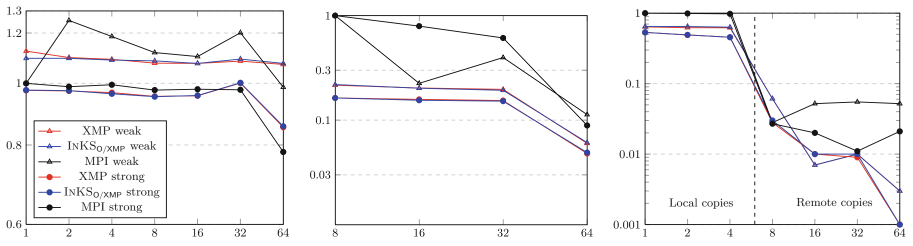
We evaluate InKS$_{\text{O/XMP}}$ and InKS$_{\text{O/Loop}}$ on Vlasov-Poisson separately as they target different optimizations and are not usable together currently. A first experiment focuses on the sequential aspects with the intra-node optimization of the v_1 advection using either InKS$_{\text{O/Loop}}$ or plain C++. A second experiment focuses on the parallel aspects with the charge density computation, the Poisson solver and a halo exchange optimized either with C/XMP or with InKS$_{\text{O/XMP}}$. The reference is the Fortran/MPI implementation from the Selalib library [13].

Distinct files for both concerns in InKS makes possible to write a unique $\text{InKS}_{\mathsf{pia}}$ algorithm and multiple versions of optimization choices. Four optimization choices are implemented based on one $\text{InKS}_{\mathsf{pia}}$ source : (1) $\text{InKS}_{\mathsf{O/XMP}}$, (2) C with XMP, (3) $\text{InKS}_{\mathsf{O/Loop}}$ and (4) plain C++. This proves, to some extent, that the separation of concerns is respected. As of now, $\text{InKS}_{\mathsf{O/XMP}}$ and $\text{InKS}_{\mathsf{O/Loop}}$ are not usable together since $\text{InKS}_{\mathsf{O/Loop}}$ relies on C++ that XMP does not support. We plan to address this limitation in the future.

For the v_1 advection, both the C++ and $\text{InKS}_{\mathsf{O/Loop}}$ optimizations of the InKS code achieve performance similar to the reference as shown in Table 2. For the parallel aspects, the $\text{InKS}_{\mathsf{O/XMP}}$ optimization offers performance similar to XMP as shown on Fig. 3. The performance is comparable to MPI on the reduction operation but MPI is faster on the Poisson solver and the halo exchanges. At the moment, it seems that XMP does not optimize local copies which slows down the Poisson solver. Besides, XMP directives used for the halo exchanges are based on MPI RMA which make the comparison with MPI Send/Receive complex. Still, MPI is much harder to program: more than 350 lines of MPI and Fortran are required to handle domain decomposition, remapping for FFT and halo exchange in Selalib vs. 50 lines in XMP and 15 in $\text{InKS}_{\mathsf{O/XMP}}$.

Table 2. Comparison between Fortran, C++ and $\text{InKS}_{\mathsf{O/Loop}}$ version of the v_1 advection on a 32^6 grid (double precision) on a single E5-2670 core with vectorization. Median of 12 executions

Version	Time/advection	GFLOP/s	% Peak core perf.
Selalib (Fortran)	4.81 s	1.12	5.36%
InKS (C++)	3.76 s	1.43	6.87%
InKS ($\text{InKS}_{\mathsf{O/Loop}}$)	3.61 s	1.49	7.16%



(a) f6D reduction (b) Poisson solver (c) Dynamic halo exchanges

Fig. 3. Weak and strong scaling for 3 parts of the Vlasov-poisson solver up to 64 nodes (1 process/node) on a 32^6 grid divided among processes (strong scaling) or 16^6 grid per process (weak scaling). Median of 10 executions.

The InKS$_{\mathsf{O}}$ family of DSLs enables the developer to specify optimization choices only while algorithmic information is extracted from InKS$_{\mathsf{pia}}$ code. This is illustrated by Listings 2, presenting the InKS$_{\mathsf{O/XMP}}$ 6D domain decomposition, and 3, presenting the XMP version. Both are equivalent, but the InKS$_{\mathsf{O/XMP}}$ expects only optimization choices parameters. Hence, one can test another memory layout, such as a different dimension ordering, by changing only a few parameters, while multiple directives must be modified in XMP. Similarly, with InKS$_{\mathsf{O/Loop}}$ (Listing 4), developers can easily test different optimization choices that would be tedious in plain C++. Since InKS$_{\mathsf{O/XMP}}$ and InKS$_{\mathsf{O/Loop}}$ are respectively usable with C and C++, InKS does not restrict the expressible optimization choices: one can still implement optimizations not handled by InKS$_{\mathsf{O}}$ in C/C++. Moreover, operations such as halo size computation or vectorization capabilities detection are automatized using the algorithm. In summary, the approach enables optimization specialists to focus on their specialty which make the development easier.

6 Conclusion and Future Works

In this paper, we have presented the InKS programming model to separate algorithm (InKS$_{\mathsf{pia}}$) and optimization choices (InKS$_{\mathsf{O}}$ & C++) and its implementation supporting two DSLs : InKS$_{\mathsf{O/Loop}}$ for loop optimizations and InKS$_{\mathsf{O/XMP}}$ for domain decomposition. We have evaluated InKS on synthetic benchmarks and on the Vlasov-Poisson solving. We have demonstrated its generality and its advantages in terms of separation of concerns to improve maintainability and portability while offering performance on par with existing approaches.

While this paper demonstrates the interest of the InKS approach, it still requires some work to further develop it. We plan to apply the InKS model on a range of different problems. We will improve the optimization DSLs; base InKS$_{\mathsf{O/Loop}}$ on existing loop optimization tools and ensure good interactions with InKS$_{\mathsf{O/XMP}}$. We also want to target different architectures to demonstrate the portability gains of the InKS approach.

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput.: Pract. Exper. **23**(2), 187–198 (2011). https://doi.org/10.1002/cpe.1631
2. Aumage, O., Bigot, J., Ejjaaouani, K., Mehrenberger, M.: INKS, a programming model to decouple performance from semantics in simulation codes. Technical report (2017). https://hal-cea.archives-ouvertes.fr/cea-01493075
3. Bailey, D.H., et al.: The NAS parallel benchmarks. Int. J. Supercomput. Appl. **5**(3), 63–73 (1991)
4. Carter Edwards, H., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014). https://doi.org/10.1016/j.jpdc.2014.07.003, http://www.sciencedirect.com/science/article/pii/S0743731514001257

5. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: Parallel Programming in OpenMP. Morgan Kaufmann Publishers Inc., San Francisco (2001)
6. Christen, M., Schenk, O., Burkhart, H.: PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: 2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 676–687. IEEE, May 2011. https://doi.org/10.1109/ipdps.2011.70
7. Cosnard, M., Jeannot, E.: Compact DAG representation and its dynamic-scheduling. J. Parallel Distrib. Comput. **58**(3), 487–514 (1999). https://doi.org/10.1006/jpdc.1999.1566, http://www.sciencedirect.com/science/article/pii/S0743731599915666
8. El-Ghazawi, T., Carlson, W., Sterling, T., Yelick, K.: UPC: distributed Shared Memory Programming (Wiley Series on Parallel and Distributed Computing). Wiley, Hoboken (2005)
9. Griebler, D., Löff, J., Fernandes, L., Mencagli, G., Danelutto, M.: Efficient NAS benchmark kernels with C++ parallel programming, January 2018
10. Höhnerbach, M., Ismail, A.E., Bientinesi, P.: The vectorization of the tersoff multi-body potential: an exercise in performance portability. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis SC 2016, pp. 7:1–7:13. IEEE Press, Piscataway (2016). http://dl.acm.org/citation.cfm?id=3014904.3014914
11. Hoque, R., Herault, T., Bosilca, G., Dongarra, J.: Dynamic task discovery in parsec: a data-flow task-based runtime. In: Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems ScalA 2017, pp. 6:1–6:8. ACM, New York (2017). https://doi.org/10.1145/3148226.3148233
12. Kamil, S.: Stencilprobe: a microbenchmark for stencil applications (2012). http://people.csail.mit.edu/skamil/projects/stencilprobe/. Accessed 25 Aug 2017
13. Kormann, K., Reuter, K., Rampp, M., Sonnendrücker, E.: Massively parallel semi-lagrangian solution of the 6D Vlasov-Poisson problem, October 2016
14. Lee, J., Sato, M.: Implementation and performance evaluation of xcalablemp: a parallel programming language for distributed memory systems. In: 2010 39th International Conference on Parallel Processing Workshops, pp. 413–420, September 2010. https://doi.org/10.1109/ICPPW.2010.62
15. Mehrenberger, M., Steiner, C., Marradi, L., Crouseilles, N., Sonnendrücker, E., Afeyan, B.: Vlasov on GPU (vog project)******. In: Proceedings ESAIM, vol. 43, pp. 37–58 (2013). https://doi.org/10.1051/proc/201343003
16. Steuwer, M., Remmelg, T., Dubach, C.: Lift: a functional data-parallel IR for high-performance GPU code generation. In: 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 74–85, February 2017. https://doi.org/10.1109/CGO.2017.7863730
17. Tang, Y., Chowdhury, R.A., Kuszmaul, B.C., Luk, C.K., Leiserson, C.E.: The Pochoir stencil compiler. In: Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures SPAA 2011, pp. 117–128. ACM (2011). https://doi.org/10.1145/1989493.1989508
18. Tanno, H., Iwasaki, H.: Parallel skeletons for variable-length lists in sketo skeleton library. In: Proceedings of the 15th International Euro-Par Conference on Parallel Processing Euro-Par 2009, pp. 666–677. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03869-3_63