The Journal of Supercomputing (2020) 76:4666–4681
https://doi.org/10.1007/s11227-019-02950-2

InKS: a programming model to decouple algorithm from optimization in HPC codes

Ksander Ejjaaouani[1] · Olivier Aumage[2] · Julien Bigot[3] · Michel Méhrenberger[4] · Hitoshi Murai[5] · Masahiro Nakao[5] · Mitsuhisa Sato[5]

Published online: 23 July 2019

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 or port to new architectures. In this paper, we propose the InKS programming model that decouples these concerns with two distinct languages: InKS_{pia} to express the simulation algorithm and InKS_{pso} for optimizations. We define InKS_{pia} and evaluate the feasibility of defining InKS_{pso} with three test languages: $\text{InKS}_{\text{o/C++}}$, $\text{InKS}_{\text{o/loop}}$ and $\text{InKS}_{\text{o/XMP}}$. We evaluate the approach on synthetic benchmarks (NAS and heat equation) as well as on a more complex example (6D Vlasov–Poisson solver). Our evaluation demonstrates the soundness of the approach as it improves the separation of algorithmic and optimization concerns at no performance cost. We also identify a set of guidelines for the later full definition of the InKS_{pso} language.

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 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

✉ Ksander Ejjaaouani
ksander.ejjaaouani@inria.fr

Extended author information available on the last page of the article

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 scientists and does not depend on the target architecture. On the other hand, optimizations form another domain of expertise and have to be adapted for a given architecture. Therefore, both algorithm and optimizations concerns are expressed within a single code. This mix impedes simulation code's maintainability and readability while hindering developer's productivity.

Many approaches have been proposed to improve this situation in the form of libraries or languages [4, 17, 19, 20]. Approaches based on automated optimization processes typically isolate the algorithmic aspects 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 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. This separation aims to improve the readability and maintainability of codes while easing portability and new optimization expression. This approach is used for common optimizations, while $\text{InKS}_{\text{o/C++}}$ is used for less common optimizations. Such a program can then be optimized using $\text{InKS}_{\text{o/XMP}}$ and $\text{InKS}_{\text{o/loop}}$, two domain-specific languages (DSLs) which ask for optimization information only. While these DSLs target some common optimizations, $\text{InKS}_{\text{o/C++}}$ can be used for less common ones.

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 and tests the InKS_{pso} approach with three optimization DSLs, $\text{InKS}_{\text{o/C++}}$, $\text{InKS}_{\text{o/loop}}$ 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 analyzes related works. Section 3 describes InKS and its implementation. Section 4 shows the use of InKS on a 6D Vlasov–Poisson solver. Section 5 evaluates our approach. Section 6 concludes the paper.

2 Related work

We now present approaches currently available to scientific programmers to help implementing simulation applications. A first widely used approach is based on imperative languages such as Fortran or C. Libraries like MPI extend this to distributed memory with message passing. Abstractions very close to the 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 [13] to support performance portability. Algorithm and optimizations are instead often tightly bound together in codes.

A second approach is offered by tools (libraries, frameworks or language extensions) that encode classical optimizations. *OpenMP* [5], *REPARA* [8] or *Kokkos* [4] supports common shared memory parallelization patterns. For example, Kokkos offers multidimensional arrays and iterators for which efficient memory mappings and iteration orders are selected independently. *UPC* [9] or XMP [17] support the partitioned global address space paradigm. For example, 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. For instance, in OpenMP or REPARA, parallel concerns are specified on top of an existing code which already carries optimization choices, such as loop order.

A third approach pushes this further with tools that automate the optimization process. For example, *PaRSEC* [12] or *StarPU* [1] supports the multitask paradigm. In StarPU, the user expresses its code as a directed acyclic graph (DAG) of tasks with data dependencies that is automatically scheduled at runtime depending on the available resources. Other examples are *SkeTo* [21] or *Lift* [19] 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 separates 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 in StarPU, for example. The algorithm remains largely interleaved with optimization choices even with this approach.

A last approach is based on DSLs that restrict the developer to the expression of the algorithm only, while optimizations are handled independently, such as *Pochoir* [20] or *PATUS* [6], DSLs for stencil problems. In Pochoir, the user specifies a stencil (computation kernel and access pattern), boundary conditions and a space–time domain, while all optimizations are handled by a compiler. These approaches ensure a very good separation of concerns. 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. On the contrary, the wider the target domain is, the less efficient optimizations are possible.

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 with (4) the possibilities of fine and manual optimizations offered by both imperative languages and dedicated tools. 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 first introduces the design of our InKS programming model, based on the use of distinct languages to express the algorithm and optimization choices separately. It then presents a prototype implementation of the model composed of InKS$_{\text{pia}}$, the algorithm language, and two InKS$_{\text{pso}}$ optimization languages. The simulation *algorithm* consists in the set of values computed, the formula used to produce them as well as the simulation inputs and outputs. *Optimization choices* include all that is not the algorithm: e.g., the computing unit selected for each computation, their ordering, the memory location 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$_{\text{pia}}$ language is used to express the algorithm with no concern for optimization choices. A compiler can automatically generate non-optimized choices from an InKS$_{\text{pia}}$ specification, mostly for testing purposes. The InKS$_{\text{pso}}$ language is used to define optimizations

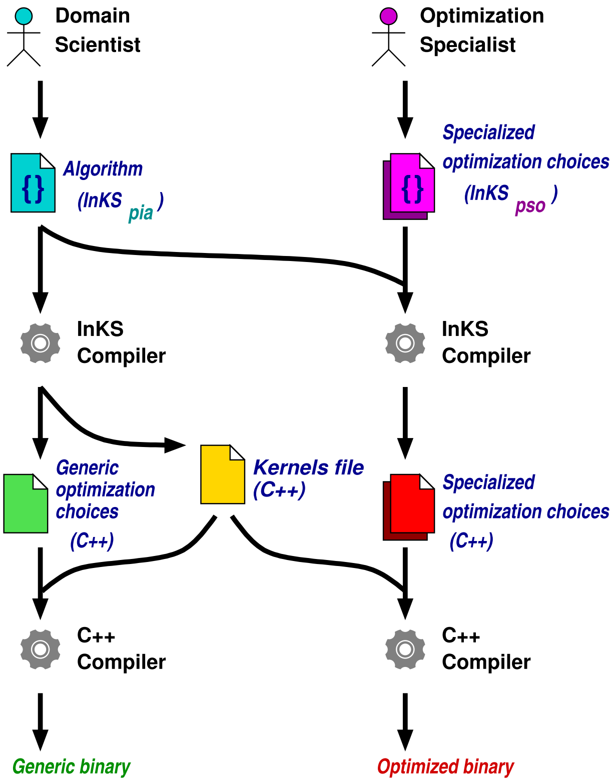


Fig. 1 InKS model

only, while other information is gathered from the InKS$_{\mathrm{pia}}$ code. Many versions of the optimization choices can be devised for a single algorithm, for example to optimize for multiple targets.

We now describe the InKS$_{\mathrm{pia}}$ language and propose three preliminary DSLs to validate strategies for the design of the InKS$_{\mathrm{pso}}$ language: InKS$_{\mathrm{o/XMP}}$ handles domain decomposition, InKS$_{\mathrm{o/loop}}$ focuses on efficient loop nest and InKS$_{\mathrm{o/C++}}$ enables to use C++ and write arbitrary complex optimization.

3.1 The InKS$_{\mathrm{pia}}$ language

In InKS$_{\mathrm{pia}}$ [2], 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 parameter 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*. Listing 1 presents a simple InKS$_{\mathrm{pia}}$ code: a 1D 3-point stencil computation. The simulation consists of one 2D logical array, `Array` (line 12), two kernels, `stencil3` (line 1) and `boundary` (line 6), and is parameterized by two integers, `X` and `T` (line 11). Line 12 specifies that the simulation starts with a subset of `Array` (every values in the space dimension, at the first time step) and expects, as output, another subset: every values in the space dimension at the last time step. To do so, it can call the `stencil3` and `boundary` kernels with a specific set of values for their integer parameters `x` and `t` (validity domain) and with `Array` as their logical array parameter, as expressed on line 15 and 16.

A InKS$_{\mathrm{pia}}$ code specifies a parameterized task graph (PTG) [7]. This graph is encoded using the polyhedron model [10]. It provides a compact representation of static control programs. This representation covers a large range of problems but imposes a few limitations. Mostly, all the problem parameters must be known at launch time, and it does, for example, not support adaptive mesh or time steps. It is still possible to express these concerns outside InKS$_{\mathrm{pia}}$ and call the InKS implementation multiple times with different parameters.

Once the algorithm is specified in InKS$_{\mathrm{pia}}$, the goal is to write the optimization choices meaning choosing a memory layout and a scheduling of the kernels. Two approaches exist to produce these choices: the automatic compiler or InKS$_{\mathrm{pso}}$. All approaches rely on the *kernel* file. This file is generated by the InKS$_{\mathrm{pia}}$ compiler which translates the InKS$_{\mathrm{pia}}$ kernels into C++ functions. The InKS$_{\mathrm{pia}}$ compiler can produce generic optimization choices with a valid but non-optimized computations scheduling and memory allocations to execute them. Scheduling and memory layouts are computed using the *Integer Set Library* [22] and recent works on modular mapping in the polyhedron model [14]. Arbitrarily complex versions of optimization choices can also be written manually in plain C++. These functions can be called from any existing code whose language supports the C calling convention. However, that approach requires information present in INKS$_{\text{pia}}$ to be repeated. The INKS$_{\text{pso}}$ DSL thus interfaces the optimization process with INKS$_{\text{pia}}$, offering the optimization specialists to specify optimization only.

```
kernel stencil3(x, t) : (
  double A(2) {in: (x-1:x+1, t-1) | out: (x, t)}
)
#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

kernel boundary(x, t) : (
  double A(2) {in: (x, t-1) | out (x, t)}
)
#CODE(C) A(x,t) = A(x, t-1); #END

public kernel inks_stencil(X, T) : (
  double Array(2) {in: (0:X, 0) | out: (0:X, T-1)}
)
#CODE(INKS)
  stencil3 (1:X-1, 1:T) : (Array),
  boundary {(0, 1:T), (X-1, 1:T)} : (Array)
#END
```

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

3.2 INKS$_{\text{pso}}$ DSL implementation

Implementing a complete optimization DSL for the INKS model is a long-term objective. In this study, we present the two DSLs: INKS$_{\text{o/XMP}}$ and INKS$_{\text{o/loop}}$. Both DSLs solely describe optimizations and get missing information from INKS$_{\text{pia}}$ code.

```
/***InKSo/XMP specification***/
#pragma xmp nodes p[nNodes]
void inks_stencil(T& Input, size_t X, size_t T){
  #pragma inks decompose % Array // allocation of "Array" algorithmic array
    (2*2, 1)  // dimension reordering, time dimension is fold by 2 and not distributed
    with t onto p // block decomposition mapped on the XMP topology

  for(int t=1; t<T; t++){
    boundary(Array, 0, t);
    for(int x=1; x<X-1; x++)
      stencil3(Array, x, t);
    boundary(Array, X-1, t);
  }
}

/***XMP + C result***/
#pragma xmp nodes p[nNodes]
#pragma xmp template t[:][:] // xmp 1d logical array
#pragma xmp distribute t[*][block] onto p
void inks_stencil(T& Input, size_t X, size_t T){
  #pragma xmp align Array[t][x] with t[t][x]
  #pragma xmp template_fix t[2][X]
  Array = (double(*)[X]) xmp_malloc(xmp_desc_of(f6d), 2, X);
  for(int t=1; t<T; t++){
    boundary(Array, 0, t);
    for(int x=1; x<X-1; x++)
      stencil3(Array, x, t);
    boundary(Array, X-1, t);
  }
}
```

Listing 2: INKS$_{\text{o/XMP}}$ optimization choices of the INKS$_{\text{pia}}$ code presented in Figure 1 and the C + XMP result

InKS$_{\text{o/XMP}}$ (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. The `inks decompose` directive supports static or dynamic allocation of logical arrays described in the algorithm. The domain size is extracted from InKS$_{\text{pia}}$ source, and the user only has to specify its mapping onto memory. As in XMP, InKS$_{\text{o/XMP}}$ supports domain decomposition mapped onto an XMP topology. In InKS$_{\text{pia}}$ code, there are no concerns for memory optimization such as dimension ordering or memory reuse. Therefore, InKS$_{\text{o/XMP}}$ supports dimension reordering and folding which consists in reusing the same memory address for subsequent indices in a given dimension. The `exchange` directive supports halo exchanges. The user specifies which dimension should be exchanged and which computational kernel will be executed after the exchange. From this information and the InKS$_{\text{pia}}$ kernel specification, the InKS$_{\text{o/XMP}}$ compiler then computes the halo size. While XMP requires halo values to be stored contiguously with the domain, InKS$_{\text{o/XMP}}$ supports a dynamic halo extension where halo values are stored in dedicated, dynamically allocated buffers to reduce memory footprint.

```
/*** stencil.iloop InKSo/Loop optimization choices file ***/
loop stencil3_loop(t, X) : stencil3 { // set "t" value
  // "Set" not specified -> loop bounds are computed, with a fixed "t"
  Order: x;  // order of the loop
}
/*** C++ result ***/
void stencil3_loop(T& Array, size_t t, size_t X){
  for (int x=0; x<X; x++)
    stencil3(Array, x, t);
}
/*** InKSo/Loop result used ***/
void inks_stencil(T& Input, size_t X, size_t T){
  for(int t=1; t<T; t++){
    boundary(Array, 0, t);
    stencil3_loop(Array, t, X);
    boundary(Array, X-1, t);
  }
}
```

Listing 3: InKS$_{\text{o/loop}}$ optimization choices of a loop from the InKS$_{\text{pia}}$ code presented in Figure 1 with its C++ result and use

InKS$_{\text{o/loop}}$ (illustrated in Listing 3) offers to specify manually loop nests for which the compiler generates plain C++ loops. Plain C++ is usable in combination with InKS$_{\text{o/loop}}$. 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$_{\text{pia}}$, but the `Set` keyword makes it possible to restrict these bounds. The `Order` keyword specifies the iteration order on the dimensions named according to the InKS$_{\text{pia}}$ code. 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 InKS$_{\text{pia}}$ code to check the validity of the loop order and generate vectorization directives whenever 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 space dimensions (x_1, x_2, x_3) and 3 in 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, only known at runtime, as described in more details in [18].

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

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 4 steps: advections in space dimensions, computation of the charge density (reduction) and electric field (Poisson solver) and advections in velocity dimensions. The algorithm of the simulation is presented in Fig. 2.

The remaining of the section presents two optimizations implemented in the *Selalib* [16] version of the 6D Vlasov–Poisson problem.

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"*). Figure 3 shows this optimization on a 2D domain. Listing 4 shows the InKS$_{\mathrm{O/XMP}}$ implementation of this strategy on 6D Vlasov–Poisson.

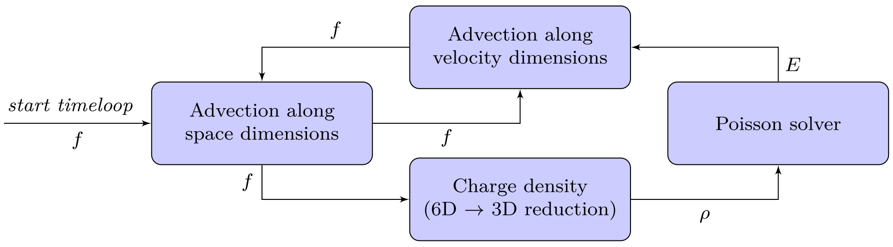

start timeloop
f
Advection along space dimensions
f
Advection along velocity dimensions
f
Charge density (6D → 3D reduction)
ρ
Poisson solver
E
f

Fig. 2 6D Vlasov–Poisson algorithm

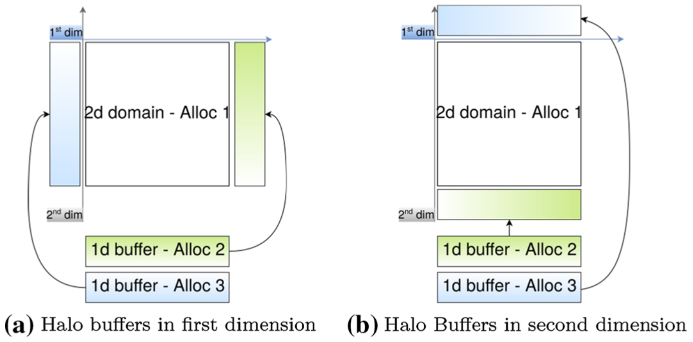


Fig. 3 Dynamic halo exchange representation on a 2D domain

```
/*** InKSo/XMP code for the f6d domain decomposition and dynamic halo exchange ***/
#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 and 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
// R and L are now allocated buffers and contain the halo values
#pragma inks exchange periodic f6d(4, advection4) to R and L
foo(R, L);
/*** Generated C+XMP code for the f6d domain decomposition ***/
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 4: 6D decomp. & halo exchange in InKS$_{\text{o/XMP}}$ and resulting code

Advections account for the main computational cost of the problem, up to for 95% 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 multiple elements at a time. Listing 5 corresponds to the InKS$_{\mathrm{o/loop}}$ implementation of these optimizations in a sequential version and presents the generated code. The Poisson solver relies on a remapping scheme, where the domain decomposition is modified between each FFT execution.

```
/*** 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)
f6d(i, j, k, l, m, n, t, step) =
    coef(n, 0) * f6d(i, j, k-1+disp(n), l, m, n, t, step-1)
  + coef(n, 1) * f6d(i, j, k+0+disp(n), l, m, n, t, step-1)
  + coef(n, 2) * f6d(i, j, k+1+disp(n), l, m, n, t, step-1)
  + coef(n, 3) * f6d(i, j, k+2+disp(n), l, m, n, t, step-1);
#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, k, i, j; // order of the loop
  Block: 16;      // blocking on the inner dimension j
  Buffer: f6d(3);    // copy the third dimension of f6d to a 1d buffer
}
```

Listing 5: A loop nest in InKS$_{\mathrm{pia}}$ optimized in InKS$_{\mathrm{o/loop}}$

5 Evaluation

This section evaluates the InKS model on the NAS benchmark, a simple stencil code and the 6D Vlasov–Poisson problem. We have implemented the algorithm of the following programs using InKS$_{\mathrm{pia}}$.

- 4 sequential NAS kernels (`IS`, `FT`, `EP` and `MG`), C++ version [11] as reference;
- finite difference 3D heat resolution (7-point stencil) ([15] as reference);
- 6D Vlasov–Poisson, using Fortran/MPI Selalib [16] as reference.

For the 3D heat equation solver, two strategies were implemented: One uses double buffering (Heat/Buf) and the other implements a cache-oblivious strategy (Heat/Obl). To evaluate the InKS programming model on Vlasov–Poisson 6D, we have conduct three experiments. The first one is based on optimization choices written in plain C++ and cover the whole simulation. Then, we evaluate InKS$_{\mathrm{o/XMP}}$ and InKS$_{\mathrm{o/loop}}$ 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$_{\mathrm{o/loop}}$

or InKS$_{\text{o/C++}}$. 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$_{\text{o/XMP}}$. Table 1 summarizes the optimization choices we have implemented. All codes are compiled with Intel 18 compiler (`-O3 -xHost`), Intel MPI 2018 and executed on the *Irene* cluster (TGCC, France) equipped with 192 GB RAM, two Skylake 8168 CPUs per node and a EDR InfiniBand interconnect. Table 2 summarizes the optimization choices we have implemented.

The NAS `CG` kernel relies on indirections not expressible in the polyhedron model of InKS$_{\text{pia}}$. Its implementation would thus have to rely on a large C++ kernel whose optimization would be mixed with the algorithm. For the same reason, the NAS `FT` kernels was only partially implemented.InKS$_{\text{pia}}$ can, however, be used to express all other NAS kernels, the 3D heat equation solver as well as the 6D Vlasov–Poisson algorithm. Even if not as expressive as C or Fortran, InKS$_{\text{pia}}$, through the polyhedron model, handles static controls programs. This 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. More specifically, it supports programs expressible as parameterized task graphs [7]: a directed acyclic graph of tasks (here kernels) in a compact representation whose dependence and scheduling are parameterized by integers fixed at the InKS entry point execution. Among others, it can express computations such as FFTs or stencils with input coordinates unknown at compile time, as in 6D Vlasov–Poisson.

InKS separates the specification of algorithm and optimization in distinct files. Multiple optimization strategies can be implemented for a single algorithm, as shown for the 3D heat equation where each relies on a specific memory layout and scheduling. Similarly, based on the InKS$_{\text{pia}}$ algorithm implemented for the `IS`, `EP` and `MG` NAS kernels, we have developed multiple optimization choices based on InKS$_{\text{o/C++}}$ and either InKS$_{\text{o/loop}}$ or InKS$_{\text{o/XMP}}$. For more complex cases such as the 6D Vlasov–Poisson problem, we were able to develop four optimization choices based on one InKS$_{\text{pia}}$ algorithm: (1) InKS$_{\text{o/XMP}}$, (2) C with XMP, (3) InKS$_{\text{o/loop}}$ and (4) InKS$_{\text{o/C++}}$. This proves, to some extent, that the separation of concerns is respected.

Finding the right metric to evaluate the easiness of writing a code is a difficult question. As illustrated in Listings 1 and 5, 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 polyhedron model used to represent the InKS$_{\text{pia}}$ code is compatible with a subset of the C language. Therefore, we could have chosen the C as the algorithm language. However, the optimization process would have suffer from such a choice. Indeed, while InKS$_{\text{pia}}$ is designed to remove optimizations through the use of DSA and fine granularity kernels, C enables its users to reuse memory and specify a total scheduling. Although it is possible to analyze the code and to retrieve the DSA form and the partial order of the code using the polyhedron model, it will be complicated for optimization specialists to find which loops can be broken or which arrays can be expended from a C code.

Concerning the specification of optimization choices, it is close to their expression in C++. Table 2 compares the GNU complexity score of InKS optimizations to

Table 1 Summary of the INKS$_{o/C++}$, INKS$_{o/loop}$ and INKS$_{o/XMP}$ optimization choices implemented based on six INKS$_{pia}$ algorithm: the 3D heat solver, 4 NAS kernels (EP, FT, MG, IS) and the Vlasov–Poisson 6D solver

Algorithm		INKS$_{o/C++}$	INKS$_{o/loop}$	INKS$_{o/XMP}$
Heat	Buf	🟩	🟩 3 versions: Same as reference, 2D and 3D cache blocking	🟩
	Obl	🟩	🟥	🟥
NAS	EP	🟩	🟥	🟩
	FT	🟩	🟥	🟥
	MG	🟩	🟩	🟥
	IS	🟩	🟩	🟥
VP6D	Initialization	🟩	🟥	🟩
	Advections	🟩	🟩 adv_v1	🟥
	Reduction	🟩	🟥	🟩
	Poisson solver	🟩	🟥	🟩

the reference code. INKS scores are slightly better, which indicates that our language is not more complex than C++. The difference comes from the extraction of the computational kernels, placed in the algorithm, which hides parts of the complexity. In addition, the use of INKS$_{o/C++}$ to write optimizations let optimization specialists reuse their preexisting knowledge of this language. Similarly writing the INKS$_{o/C++}$ version of optimization choices for the INKS$_{pia}$ version of Vlasov–Poisson 6D is close to the expression to the same optimization in Fortran, in the reference version. These considerations should not hide the fact that some information has to be specified in both the INKS$_{pia}$ and INKS$_{o/C++}$ files with this approach leading to more code overall.

Table 2 Execution time of the INKS$_{o/C++}$, INKS$_{o/loop}$ and INKS$_{o/XMP}$ implementations of the sequential NAS benchmark, class B—time/iteration of the 3D heat equation (7-point stencil), size (1024^3)—time/iteration of the INKS$_{o/C++}$ and Fortran implementation of the sequential 6D Vlasov–Poisson, size (32^6)

Benchmark	Execution time (second)			Complexity	
	Reference	INKS$_{o/C++}$	Rel. dev.	Ref.	INKS
NAS/FT	49.16 (±0.16)	44.00 (±0.05)	*11.66%*	6	5
NAS/IS	1.91 (±0.00)	1.90 (±0.01)	0.53%	55	52
NAS/MG	4.61 (±0.02)	4.29 (±0.01)	*7.46%*	20	12
NAS/EP	53.52 (±0.02)	55.35 (±0.37)	*−3.30%*	19	19
Heat/Buf	2.76 (±2.78)	2.93 (±3.53)	*−6.22%*	5	3
Heat/Obl	1.20 (±1.16)	1.18 (±0.96)	1.81%	22	13
VP6D	25.92 (±0.21)	21.92 (±0.16)	*18.28%*	N/A	N/A

Benchmark	Reference (C++)	INKS$_{o/loop}$	Rel. dev.	Reference (Fortran)
NAS/IS	1.91 (±0.00)	1.91 (±0.01)	0.00%	N/A
NAS/MG	4.61 (±0.02)	4.32 (±0.02)	*6.84%*	N/A
VP6D	4.18 (±0.05)	4.23 (±0.06)	−1.10%	3.63 (±0.09)

Benchmark	Reference	INKS$_{o/XMP}$	Efficiency (# cores)
NAS/EP	53.52 (±0.02)	14.00 (±0.13)	70.5% (4 cores)

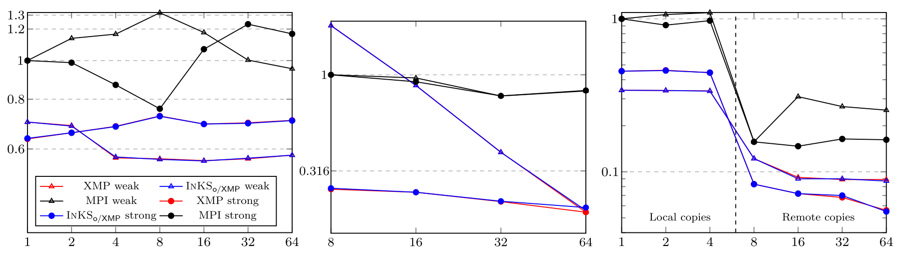
Median and standard deviation of 10 executions-GNU Complexity score of the implementation

On the contrary, InKS$_{\text{o/XMP}}$ and InKS$_{\text{o/loop}}$ enable the developer to specify optimization choices only, while algorithmic information is extracted from InKS$_{\text{pia}}$ code. This is illustrated in Listing 2 presenting the InKS$_{\text{o/XMP}}$ 6D domain decomposition and the XMP result. Both are equivalent, but the InKS$_{\text{o/XMP}}$ 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. Moreover, using the domain decomposition directive offered by InKS$_{\text{o/XMP}}$ and XMP code, one can derive a simple parallel code from a InKS$_{\text{pia}}$ algorithm, as we did with the 3D Heat solver. Note that since InKS$_{\text{o/XMP}}$ is a wrapper for XMP using InKS$_{\text{pia}}$ to retrieve some information, it is usable in any codes that can be expressed using InKS$_{\text{pia}}$ and optimized with XMP. Similarly, with InKS$_{\text{o/loop}}$ (Listing 5), developers can easily test different optimization choices that would be tedious in plain C++. This is what we did with the 3D Heat solver based on the double buffering technique. As shown in Fig. 1, using InKS$_{\text{o/loop}}$, we have implemented 3 versions of the loops: the same as reference, one based on a 2D cache blocking and a last one using a 3D cache blocking by modifying almost nothing in the InKS$_{\text{o/loop}}$ code. Since InKS$_{\text{o/XMP}}$ and InKS$_{\text{o/loop}}$ are, respectively, usable with C and C++, InKS does not restrict the expressible optimization choices: One can still implement optimizations not handled by our DSLs in C/C++. Moreover, operations such as halo size computation or vectorization capabilities detection are automatized using the InKS$_{\text{pia}}$ code. In summary, the approach enables optimization specialists to focus on their specialty which make the development easier.

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 matches or improves upon the reference as presented in Table 2. Investigation has shown that Intel ICC 18 does not vectorize properly the reference versions of `MG` and `FT`. The use of the Intel `ivdep` directive as done on the InKS versions leads to slightly better performance.

For the full Vlasov–Poisson 6D problem, InKS$_{\text{pia}}$ and C++ enable us to implement the same complex optimization strategies written in the reference version. Therefore, our implementation match the reference in terms of performance, as shown in Table 2. For the v_1 advection, both the InKS$_{\text{o/C++}}$ and InKS$_{\text{o/loop}}$ optimizations of the InKS code achieve performance similar to the reference as shown in Table 2. For the parallel aspects, the InKS$_{\text{o/XMP}}$ optimization offers performance similar to XMP as shown in Fig. 4. MPI is faster on all cases compared to both XMP and InKS$_{\text{o/XMP}}$. At the moment, it seems that XMP does not optimize local copies which slow down the Poisson solver. Besides, some XMP directives are based on MPI RMA which makes 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, while 50 lines in XMP and 15 in InKS$_{\text{o/XMP}}$.

However, InKS$_{\text{pia}}$, InKS$_{\text{o/loop}}$ and InKS$_{\text{o/XMP}}$ have limitations. As mentioned earlier, InKS$_{\text{pia}}$ cannot be used to express non-static control program, such as the `CG` or the full `FT` NAS kernel. It also makes more complex the optimization of arrays



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

Fig. 4 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

passed in parameters (as input or output). INKS$_{o/loop}$ offers very limited option, making it unusable with complex loops such as the one in the 3D heat solver optimized using the cache-oblivious strategy. Similarly, loop nests calling multiple INKS$_{pia}$ computational kernels cannot be expressed using INKS$_{o/loop}$. Moreover, it cannot respect the constraints imposed by a specific memory mapping and offers only a few optimization. Adding new optimization strategies would require to add new keywords, such as unrolling and loop fusion. INKS$_{o/XMP}$ offers memory allocations but no controls on the access to this memory, letting optimization specialists in charge of the good use of this memory. Besides, it is usable only with XMP.

Although we want to address some issues in the INKS$_{pia}$ language, INKS$_{o/loop}$ and INKS$_{o/XMP}$ were preliminary tests before a real INKS$_{pso}$ implementation. The goal was to propose a way to express optimizations and retrieving the algorithmic information in INKS$_{pia}$ code. These two DSLs enable us to highlight a set of guidelines for the full definition of the INKS$_{pso}$ language. First, this definition must be based on the concepts that INKS$_{pso}$ must express, i.e., the optimizations related to memory (allocations and layouts) and to computations scheduling. With INKS$_{o/XMP}$, we tried some tests on the memory aspects, providing a directive to allocate a logical array and to reorder its dimensions. INKS$_{o/loop}$ focuses on computations scheduling by adding strong constraint on the order of the computations. Secondly, in order to express these optimization concepts only, it must be bound to its algorithmic counterpart, express in INKS$_{pia}$. In INKS$_{o/XMP}$, the domain decomposition directive makes reference directly to logical arrays described in INKS$_{pia}$, making possible the computation of the size of each dimension and the halo size depending on the data being accessed. In INKS$_{o/loop}$, the `Order` keyword refers to the structuring variable of a computational kernel defined in INKS$_{pia}$. Finally, we must work on the mean to express these optimization concepts with the most generality. For this part, we think that working with existing tools is probably mandatory for most complex strategy, such as XMP for the domain decomposition. Thus, using another existing tool for computation scheduling would make o/loop more general. Similarly, a declarative language such as INKS$_{o/loop}$ may not be the best strategy: To handle the interaction between memory and computation, an imperative language could be easier.

6 Conclusion and future works

In this paper, we have presented the INKS programming model to separate the algorithm and the optimization choices and its implementation supporting two DSLs: INKS$_{o/loop}$ for loop optimizations and INKS$_{o/XMP}$ for domain decomposition. We have evaluated INKS on synthetic benchmarks and on the Vlasov–Poisson solving. We have demonstrated its generality and advantages in terms of separation of concerns to improve maintainability and portability while offering performance on a par with existing approaches.

While this paper demonstrates the interest of the INKS model, it still requires some work to further develop it. We will improve the optimization DSLs; base INKS$_{o/loop}$ on existing tools and ensure interactions with INKS$_{o/XMP}$. This will be done within the scope of the development of a complete INKS$_{pso}$ DSL enabling its users to manage memory placement and computations scheduling. This planed DSL will truly separate algorithm from optimization choices in static programs, while its compiler could offer static analysis of the program to ensure correctness. We also want to target different architectures to demonstrate the portability gains of the INKS model.

References

1. Augonnet C, Thibault S, Namyst R, Wacrenier PA (2011) StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr Comput Pract Exper 23(2):187–198. https://doi.org/10.1002/cpe.1631
2. Aumage O, Bigot J, Ejjaaouani K, Mehrenberger M (2017) InKS, a programming model to decouple performance from semantics in simulation codes. Technical report, Inria
3. Bailey DH, Barszcz E, Barton JT, Browning DS, Carter RL, Dagum L, Fatoohi RA, Frederickson PO, Lasinski TA, Schreiber RS, Simon HD, Venkatakrishnan V, Weeratunga SK (1991) The NAS parallel benchmarks. Int J Supercomput Appl 5(3):63–73. https://doi.org/10.1177/109434209100500306
5. Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R (2001) Parallel programming in OpenMP. Morgan Kaufmann, Los Altos
6. Christen M, Schenk O, Burkhart H (2011) PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: Parallel and distributed processing symposium (IPDPS) 2011, IEEE. https://doi.org/10.1109/ipdps.2011.70
7. Cosnard M, Jeannot E (1999) Compact dag representation and its dynamic scheduling. J Parallel Distrib Comput 58(3):487–514. https://doi.org/10.1006/jpdc.1999.1566
8. Danelutto M, García J, Miguel Sanchez L, Sotomayor R, Torquati M (2016) Introducing parallelism by using REPARA C++11 attributes. pp 354–358. https://doi.org/10.1109/PDP.2016.115
4. Edwards HC, Trott CR, Sunderland D (2014) Kokkos. J Parallel Distrib Comput 74(12):3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003
9. El-Ghazawi T, Carlson W, Sterling T, Yelick K (2005) UPC: distributed shared memory programming. Wiley, London
10. Feautrier P, Lengauer C (2011) Polyhedron model. Springer, London. https://doi.org/10.1007/978-0-387-09766-4_502
11. Griebler D, Loff J, Mencagli G, Danelutto M, Fernandes LG (2018) Efficient NAS benchmark kernels with c++ parallel programming. In: 2018 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). https://doi.org/10.1109/PDP2018.2018.00120
12. Hoque R, Herault T, Bosilca G, Dongarra J (2017) Dynamic task discovery in PaRSEC: a data-flow task-based runtime. In: 8th workshop on latest advances in scalable algorithms for large-scale systems, ACM
13. Höhnerbach M, Ismail AE, Bientinesi P (2016) The vectorization of the Tersoff multi-body potential: an exercise in performance portability. In: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE

14. Isoard A (2016) Extending polyhedral techniques towards parallel specifications and approximations. Ph.D. thesis, École doctorale en Informatique et Mathématiques de Lyon
15. Kamil S (2012) StencilProbe: a microbenchmark for stencil applications. Accessed 25 Aug 2017
16. Kormann K, Reuter K, Rampp M (2019) A massively parallel semi-Lagrangian solver for the six-dimensional Vlasov–Poisson equation. Int J High Perform Comput Appl. https://doi.org/10.1177/1094342019834644
17. Lee J, Sato M (2010) Implementation and performance evaluation of XcalableMP: a parallel programming language for distributed memory systems. In: International Conference on Parallel Processing Workshops
18. Mehrenberger M, Steiner C, Marradi L, Crouseilles N, Sonnendrucker E, Afeyan B (2013) Vlasov on GPU (VOG project). In: ESAIM: Proceedings of 43. https://doi.org/10.1051/proc/201343003
19. Steuwer M, Remmelg T, Dubach C (2017) LIFT: a functional data-parallel IR for high-performance GPU code generation. In: 2017 IEEE/ACM international symposium on code generation and optimization (CGO)
20. Tang Y, Chowdhury RA, Kuszmaul BC, Luk CK, Leiserson CE (2011) The Pochoir stencil compiler. In: 23rd symposium on parallelism in algorithms and architectures, ACM, SPAA '11. https://doi.org/10.1145/1989493.1989508
21. Tanno H, Iwasaki H (2009) Parallel skeletons for variable-length lists in SkeTo skeleton library. In: Proceedings of the 15th International Euro-Par Conference on Parallel Processing, Springer, Euro-Par '09. https://doi.org/10.1007/978-3-642-03869-3_63
22. Verdoolaege S (2010) isl: an integer set library for the polyhedral model. In: Fukuda K, Hoeven J, Joswig M, Takayama N (eds) Mathematical software—ICMS 2010. Springer, New York

Affiliations

Ksander Ejjaaouani[1] · Olivier Aumage[2] · Julien Bigot[3] · Michel Méhrenberger[4] · Hitoshi Murai[5] · Masahiro Nakao[5] · Mitsuhisa Sato[5]

Olivier Aumage
olivier.aumage@inria.fr

Julien Bigot
julien.bigot@cea.fr

Michel Méhrenberger
mehrenbe@math.unistra.fr

Hitoshi Murai
h-murai@riken.jp

Masahiro Nakao
masahiro.nakao@riken.jp

Mitsuhisa Sato
msato@riken.jp

1 Maison de la Simulation, CEA, CNRS, Univ. Paris-Sud, UVSQ, Université Paris-Saclay, Inria, Gif-sur-Yvette, France

2 Inria, LaBri, Bordeaux, France

3 Maison de la Simulation, CEA, CNRS, Univ. Paris-Sud, UVSQ, Université Paris-Saclay, Gif-sur-Yvette, France

4 Université de Marseille, Marseille, France

5 Riken CCS, Kobe, Japan