2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC) | 978-1-6654-6021-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/P3HPC56579.2022.00012

Performance portable Vlasov code with C++ parallel algorithm

Yuuichi Asahi
CCSE
Japan Atomic Energy Agency
Chiba, Japan
asahi.yuichi@jaea.go.jp

Thomas Padioleau
Université Paris-Saclay, UVSQ, CNRS, CEA
Maison de la Simulation
Gif-sur-Yvette, France

Guillaume Latu
DES/IRESNE/DEC
CEA
St.Paul-lez-Durance, France

Julien Bigot
Université Paris-Saclay, UVSQ, CNRS, CEA
Maison de la Simulation
Gif-sur-Yvette, France

Virginie Grandgirard
IRFM
CEA
St.Paul-lez-Durance, France

Kevin Obrejan
IRFM
CEA
St.Paul-lez-Durance, France

***Abstract*—This paper presents the performance portable implementation of a kinetic plasma simulation code with C++ parallel algorithm to run across multiple CPUs and GPUs. Relying on the language standard parallelism `stdpar` and proposed language standard multi-dimensional array support `mdspan`, we demonstrate that a performance portable implementation is possible without harming the readability and productivity. We obtain a good overall performance for a mini-application in the range of 20 % to the Kokkos version on Intel Icelake, NVIDIA V100, and A100 GPUs. Our conclusion is that `stdpar` can be a good candidate to develop a performance portable and productive code targeting the Exascale era platform, assuming this approach will be available on AMD and/or Intel GPUs in the future.**

***Index Terms*—Intel Icelake; NVIDIA V100; NVIDIA A100; AMD MI100; Semi-Lagrangian; Kokkos; OpenMP; C++17 Parallel algorithm; C++23 mdspan;**

I. Introduction

In 2022, the Exascale supercomputing era has finally come [1]. The Frontier system at Oak Ridge National Laboratory (ORNL) has achieved an HPL score of 1.102 Exaflop/s. The top 10 world fastest supercomputers in the Top 500 list (2022 June) include three AMD GPU machines and four NVIDIA GPU machines. Other machines employ multi-core CPUs. The Intel GPU based Exascale machine Aurora will be installed in 2023. Accordingly, the Exascale supercomputers have a different kind of architectures.

The more architectures, the less appealing the architecture specific implementations are, since the lifetime of an application dedicated to a specific architecture is shortened to the lifetime of the architecture. In order to minimize programming efforts to work on these divergent architectures, application developers seek for programming models that allow a single codebase to run efficiently on many architectures, and provide performance, portability, and productivity (P3). There are multiple approaches to provide performance portability over CPUs and GPUs such as library-based, directive-based, and language-based. As well as the divergent architectures, the performance portable approaches are also divergent. One school relies on directives such as OpenMP [2] and OpenACC [3]. Another school relies on libraries to build abstractions like Kokkos [4], [5], RAJA [6], and SYCL [7].

Several studies compared the capabilities and effectiveness of these frameworks in terms of practical performance portability in applications [8]–[11]. From the user point of view, we need to carefully choose the framework that fits best for the code development and maintenance strategy of a group, which is already a demanding task. Fortunately, a potential game changer may already exist: an approach integrated in the C++ language standard. A recent study shows promising performance of a lattice Boltzmann method based fluid simulation code over CPUs and NVIDIA GPUs with C++ parallel algorithms (referred to as `stdpar` in the remaining of the paper) [12]. If this approach is effective for a large variety of applications, we may prefer it to others since the lifetime of an application can technically be elongated to the lifetime of the language which should be longer than that of libraries, directives, and architectures.

In this work, we evaluate the capability of C++ parallel algorithm as a performance portable framework. Through the comparison with other frameworks like Kokkos and OpenMP, we also discuss how competitive this approach is. As a case study, we evaluate the performance of a kinetic plasma simulation mini-application [13], [14] developed to explore a suitable performance portable implementation for the 5D (3D space and 2D velocity space) plasma turbulence code GYSELA [15]. We have developed both single process [13] and MPI [14] versions of this mini-application which encapsulate the key features of GYSELA. In the present work, we implement the mini-app with `stdpar` and `mdspan` [16] and compare the performance, portability, and productivity with Kokkos, OpenMP, and Thrust [17] implementations. `mdspan` is proposed for integration in C++23 to support high dimensional array inspired by the developments of the Kokkos library. We also evaluate a performance portability metric [18], [19] on costly kernels in the mini-app.

2831-3909/22/$31.00 ©2022 IEEE
DOI 10.1109/P3HPC56579.2022.00012

This paper is organized as follows. Section II describes the testbed and details the implementations in each programming model. In Section III, we provide the basic benchmarks using a 3D heat equation solver. In Section IV, we evaluate and compare the performance of our kinetic plasma mini-applications with multiple implementations. Section V presents a short discussion about the performance, portability, and productivity achieved with `stdpar` and `mdspan` implementation. The results obtained are summarized in Section VI.

II. Comparison of frameworks

In this paper, we employ Thrust, Kokkos, OpenMP, and `stdpar` as programming models and compare the performance, portability and productivity. To develop a performance portable and productive (P3) code, two features are most relevant: data structures and parallelization methods. The best suited data structure for HPC in C++ is `mdspan` [16], which will support the high dimensional array with the data layout abstraction. For parallelization, we rely on the C++17 parallel algorithm (`stdpar`) which can be used for GPU codes with NVIDIA HPC SDK. In this section, we demonstrate our coding strategy with `stdpar` and `mdspan` and discuss the analogy to the Kokkos implementation. For `mdspan`, we use the reference implementation [16] (SHA 3aad018) on Github page [20].

A. Settings for performance measurements

The source codes used in this study are publicly available on GitHub [21]. All the measurements have been conducted 10 times, and the averaged values are used as performance data. Performance has been measured on Wisteria and SGI8600 supercomputers. For multi-core CPUs, we use Intel Xeon Platinum 8360Y (referred to as Icelake). On Icelake, we disable Hyperthreading and use all the hardware threads (=36). For GPUs, NVIDIA V100 [22] and A100 [23], and AMD MI100 [24] are employed. Kokkos, Thrust, and OpenMP versions run across all of these architectures, whereas v is unavailable on AMD MI100. For MPI, we map 1 MPI process on each device, i.e., 1 MPI process to a single socket on Icelake and 1 MPI process to a GPU card on other devices. Table I summarizes the devices and compilers used in the present work. Since our AMD testbed has only one node (4 MI100 GPUs), we cannot measure and compare weak and/or strong scalability of mini-apps. Accordingly, we focus on the intra-node performance portability in the present work.

B. Data structure

The most comprehensive data structure for HPC usage may be the View abstraction offered in the Kokkos programming model [4], [5]. The kokkos View consists of `Memory Space`, `Memory Layout`, and `Memory Traits` abstractions. It also encapsulates the data allocator dedicated to a specific hardware. These are essential features to achieve performance portability over large variety of devices.

In `stdpar`, we can instead rely on `mdspan` [16] which is highly influenced by the Kokkos View. Although the `Memory Layout` abstraction is available in `mdspan`, C++ does not have a concept of `Memory Space` nor `Memory Traits`. In addition, `mdspan` is not equipped with a data allocator. Thus, we rely on virtual unified memory (managed memory) which can be accessed both from host and device. In the present study, we employ `std::vector` as a data allocator for simplicity. Let us think of 3D (shaped with (nx, ny, nz)) arrays u and un with `Kokkos::View` and `mdspan` as shown in Listings 1 and 2. In Kokkos, the Layout abstraction is established with `Kokkos::DefaultExecutionSpace` (line 1-2 in Listing 1), where the Fortran Layout (`LayoutLeft`) is used for GPUs and the C layout (`LayoutRight`) is used for CPUs as a default. In `stdpar`, the Layout can be specified with the template argument of `mdspan` (line 2-3 in Listing 2). We rely on unsigned 64bit dynamic extents over all dimensions in order to control the problem size at runtime (line 1). We allocate data with `std::vector`, whose data pointer is then viewed through `mdspan` (line 4-5).

Listing 1. 3D array in Kokkos

```
1 using execution_space = Kokkos::DefaultExecutionSpace;
2 using RealView3D = Kokkos::View<double***, execution_space>;
3 RealView3D u("u", nx, ny, nz), un("un", nx, ny, nz);
```

Listing 2. 3D array in `stdpar`

```
1 using extents = stdex::dextents<size_t, 3>;
2 using layout = stdex::layout_left;
3 using RealView3D = stdex::mdspan<double, extents, layout>;
4 std::vector<double> _u(nx*ny*nz), _un(nx*ny*nz);
5 RealView3D u(_u.data(), {nx, ny, nz}), un(_un.data(), {nx, ny, nz});
```

In Thrust, we employ the `thrust::device_vector` to allocate data on GPUs. For the OpenMP implementation, we introduce an in house View class whose accessor is available inside the accelerated region. To avoid the extra data creation, we introduced the `omp_attach` functionality [26], which attaches a device address to a device pointer without creating an entry on the OpenMP present table. This function is used to swap View instances, for example.

C. Parallel operations

For scientific computing, parallel iteration and reduction are frequently used. Kokkos offers `parallel_for` and `parallel_reduce` for these parallel operations. In `stdpar`, there are multiple functions available to achieve these parallel operations. In the present work, we employ `std::for_each_n` and `std::transform_reduce` with counting iterators for flexibility. We always use `std::execution::par_unseq` execution policy which informs the compiler that the operation is both parallelizable and vectorizable (unsequenced).

We demonstrate the examples of parallel operations in a 3D heat equation solver. Algorithm 1 shows one time step of the 3D heat equation solver. We assume periodic boundary conditions in each dimension. For validation, we compute the L2 norm of the difference between the numerical and the analytical solutions as $\|u^{\mathrm{numerical}} - u^{\mathrm{analytical}}\|_2$.

1) Parallel iteration: Listings 3 and 4 show the Kokkos and `stdpar` implementation of the 3D heat solver respectively. Both in the Kokkos and `stdpar` versions, we define parallel operations in functors whose operators are decorated with

TABLE I
HARDWARE DESCRIPTION FOR ONE PROCESSOR. THERMAL DESIGN POWER (TDP) IS EXTRACTED FROM VENDORS DATA-SHEETS [22]–[25].

Processor	Intel Xeon Platinum 8360Y (Icelake)	NVIDIA V100 (V100)	NVIDIA A100 (A100)	AMD MI100 (MI100)
Number of cores (FP64)	36	2560	3456	-
Shared Cache [MB]	54	6	40	8
Peak performance [GFlops]	2764.8	7800	9700	11500
Peak B/W [GB/s]	204.8	900	1555	1230
B/F ratio	0.074	0.115	0.160	0.107
SIMD width	512 bit	-	-	-
Warp/wavefront size	-	32	32	64
TDP [W]	250	300	400	300
Manufacturing process [nm]	10	12	7	7
Year	2021	2017	2020	2020
Compilers (Thrust)	CUDA/11.2.152	CUDA/11.4.100	CUDA/11.2.152	rocm 5.2.2
Compilers (Kokkos)	ICC 2021.2.0	CUDA/11.4.100	CUDA/11.2.152	rocm 5.2.2
Compilers (OpenMP)	ICC 2021.2.0	NVIDIA HPC SDK 22.3	NVIDIA HPC SDK 22.5	rocm 5.2.2
Compilers (`stdpar`)	NVIDIA HPC SDK 22.5	NVIDIA HPC SDK 22.3	NVIDIA HPC SDK 22.5	N/A

Algorithm 1 One time step of heat3d solver

1: **Input:** u^n, **Output:** u^{n+1}
2: Halo exchange on u^n (P2P communications or swap)
3: 3D heat equation for Δt: $u^n \rightarrow u^{n+1}$ (forward difference in time and central difference in space)

`KOKKOS_INLINE_FUNCTION` (lines 8-9 in Listing 3) and `MDSPAN_FORCE_INLINE_FUNCTION` (lines 8-9 in Listing 4). `MDSPAN_FORCE_INLINE_FUNCTION` is unnecessary for `stdpar` but it is needed for CUDA, HIP, or Thrust, and thus we added this macro in Listing 4.

Listing 3. 3D Heat equation with Kokkos

```
1  struct heat_functor {
2    RealView3D u_, un_;
3    heat_functor(Config &conf, RealView3D &u, RealView3D &un) {
4      u_ = u; un_ = un;
5      ... // initialization
6    }
7
8    KOKKOS_INLINE_FUNCTION
9    void operator()(const int ix, const int iy, const int iz) const {
10     un_(ix, iy, iz) = u_(ix, iy, iz)
11                     + coef_ * ( u_(ix+1, iy, iz) + u_(ix-1, iy, iz)
12                               + u_(ix, iy+1, iz) + u_(ix, iy-1, iz)
13                               + u_(ix, iy, iz+1) + u_(ix, iy, iz-1)
14                               - 6. * u_(ix, iy, iz) );
15   }
16 };
17 MDPolicy<3> policy3d({0, 0, 0}, {nx, ny, nz}, {TX, TY, TZ});
18 auto heat_eq = heat_functor(conf, u, un);
19 Kokkos::parallel_for(policy3d, heat_eq);
```

Kernels are parallelized with `Kokkos::parallel_for` in Kokkos (line 19 in Listing 3) and `std::for_each_n` in `stdpar` (lines 18-25 in Listing 4). In Kokkos, we can simply define the 3D iteration policy or `MDRangePolicy` which abstracts the multidimensional loops. This policy activates tiling for CPUs and 3D thread mapping for GPUs, which can sometimes improve the performance [5], [13]. Here, $(\mathrm{TX}, \mathrm{TY}, \mathrm{TZ})$ are tile sizes in (x, y, z) directions. In `stdpar`, we need to map 1D index into 3D indices by hand as shown in Listing 4 (lines 21-23). On GPUs, the loop iterates from the leftmost index for contiguous memory accesses. This could be simplified with the Cartesian product iterator support [27] proposed for inclusion in C++.

Listing 4. 3D Heat equation with `stdpar`

```
1  struct heat_functor {
2    RealView3D u_, un_;
3    heat_functor(Config &conf, RealView3D &u, RealView3D &un) {
4      u_ = u; un_ = un;
5      ... // initialization
6    }
7
8    MDSPAN_FORCE_INLINE_FUNCTION
9    void operator()(const int ix, const int iy, const int iz) const {
10     un_(ix, iy, iz) = u_(ix, iy, iz)
11                     + coef_ * ( u_(ix+1, iy, iz) + u_(ix-1, iy, iz)
12                               + u_(ix, iy+1, iz) + u_(ix, iy-1, iz)
13                               + u_(ix, iy, iz+1) + u_(ix, iy, iz-1)
14                               - 6. * u_(ix, iy, iz) );
15   }
16 };
17 auto heat_eq = heat_functor(conf, u, un);
18 std::for_each_n(std::execution::par_unseq,
19                 counting_iterator(0), nx*ny*nz,
20                 [=](const int idx) {
21                   const int ix = idx % nx;
22                   const int iy = (idx / nx) % ny;
23                   const int iz = (idx / nx) / ny;
24                   heat_eq(ix, iy, iz);
25                 });
```

2) Parallel reduction: Listings 5 and 6 show the Kokkos and `stdpar` implementations to compare the numerical and analytical solutions of the 3D heat solver. A parallel kernel can also be defined by a lambda function. We need to add `KOKKOS_LAMBDA` (line 2 in listing 5) in Kokkos, whereas in `stdpar` we use "[=]" (capture by value) as shown in line 1 in listing 6). The kernels are parallelized with `Kokkos::parallel_reduce` (line 7 in listing 5) and `std::transform_reduce` (lines 6-15 in listing 6).

Similar to the Kokkos Views, `mdspan` keeps the meta data only without the actual data on memory. Thanks to this feature, capturing `mdspans` as well as raw pointers only impose negligible overheads. We can therefore just leverage

Listing 5. Parallel reduction with Kokkos

```
auto L2norm_kernel =
  KOKKOS_LAMBDA (int ix, int iy, int iz, double& sum) {
    auto diff = un(ix, iy, iz) − u(ix, iy, iz);
    sum += diff ∗ diff;
};
double l2norm = 0;
Kokkos::parallel_reduce(policy3d, L2norm_kernel, l2norm);
```

Listing 6. Parallel reduction with `stdpar`

```
auto L2norm_kernel = [=] (const int ix, const int iy, const int iz) {
  auto diff = un(ix, iy, iz) − u(ix, iy, iz);
  return diff ∗ diff;
};
const int n = nx ∗ ny ∗ nz;
auto l2norm = std::transform_reduce(std::execution::par_unseq,
                    counting_iterator(0), counting_iterator(0)+n,
                    0,
                    std::plus<double>(),
                    [=](const int idx) {
                      const int ix = idx % nx;
                      const int iy = (idx / nx) % ny;
                      const int iz = (idx / nx) / ny;
                      return L2norm_kernel(ix, iy, iz);
                    });
```

the benefits of `mdspan` such as multi-dimensional indexing and layout abstraction. The former contributes to readability and the latter sometimes improves performance portability.

D. Brief summary of our implementations

In this subsection, we briefly describe our implementations with Thrust, Kokkos, OpenMP and `stdpar` backends as summarized in Table II. For Thrust, Kokkos, and `stdpar`, we can define a parallel operation either with a functor or a lambda. For these implementations, we use a single codebase except for dedicated math libraries.

1) Thrust implementation: On MI100, we rely on rocThrust [28], which is the Thrust library [17] ported to HIP/ROCm platform. We set the macro `THRUST_DEVICE_SYSTEM` to `THRUST_DEVICE_SYSTEM_OMP` on CPUs.

2) Kokkos implementation: We configure Kokkos with `-DKokkos_ENABLE_OPENMP=On` for Icelake, `-DKokkos_ENABLE_CUDA=On` for V100 and A100, and `-DKokkos_ENABLE_HIP=On` for MI100.

3) OpenMP implementation: We use OpenMP offload and non-offload directives for GPUs and CPUs through preprocessor conditional expressions depending on the device type. The unstructured mapping [2] of the raw pointer members of a View instance is performed in constructor of our in-house View class. All the n dimensional loops are parallelized with "#pragma omp target teams distribute parallel for simd collapse(n)" for GPUs. For CPUs, we insert the "#pragma omp simd" along the innermost loop and parallelize the outermost loops with "#pragma omp parallel for schedule(static) collapse(2)".

4) `stdpar` implementation: We use the same code and only change compilation flags to target GPUs or CPUs. We use `-stdpar=gpu` and `-stdpar=multicore` options for GPUs and CPUs, respectively. If `stdpar` implementation does not allow the coexistence of parallelization for CPUs and GPUs in a single code (all the parallel regions are performed either on CPUs or GPUs), it can be regarded as a weakness.

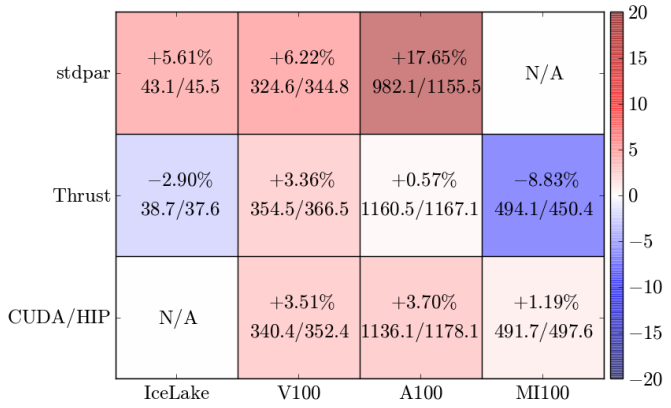

Fig. 1. Performance overhead with `mdspan` in the 3D heat solver on multiple platforms. The numbers in each grid represent the `mdspan` overheads relative to the raw pointer versions, followed by the memory bandwidth [GB/s] with `mdspan` version over the memory bandwidth [GB/s] with the raw pointer version. The x and y axes represent the device and programming model, respectively. N/A denotes an invalid combination.

III. Basic benchmarks

For the basic benchmarks, we have implemented a 3D heat equation solver with multiple implementations as in Listings 3 and 4 and compared the performance. The overhead due to the use of `mdspan` instead of raw pointers is also evaluated. Since we are mainly interested in the performance portability, we employ Thrust, Kokkos, OpenMP, and `stdpar` as programming models in the present paper. In this section, however, we also show the performance with CUDA and HIP for comprehensive understanding of the basic aspects of `stdpar` and `mdspan`. Performance evaluation with other competitive performance portable layers like RAJA [6] and SYCL [7] is left for future work.

A. Single processor performance of the 3D heat solver

To investigate the basic performance of `stdpar`, we evaluate the performance of the 3D heat solver (7 stencil computations) described in algorithm 1. The problem size is fixed as $(N_x, N_y, N_z) = (512, 512, 512)$ with 1000 iterations.

Firstly, we have investigated the overhead of `mdspan` by comparing performance with and without `mdspan`. This is similar to the stencil benchmark performed in the original `mdspan` paper [16], where it is reported that the performance overhead from `mdspans` against raw pointers is less than 10%.

Figure 1 shows performance overhead in our 3D heat solver with `mdspan`. For Thrust, we find almost zero or negative overheads. For `stdpar`, we find the maximum overhead of 17.65% on A100. If we use `mdspan` and map thread blocks $(32, 8, 1)$ in CUDA, we find the maximum performance overhead of 22.41% on A100. This may be relevant to low performance of the Kokkos version on A100 with the same thread block mapping as the CUDA version (see Fig. 2). A potential cause of the overhead is the extent type of `mdspan`, where we use 64 bit unsigned integers. In the raw pointer case, we use 32 bit signed integers. Thus, we may suffer from

TABLE II
IMPLEMENTATIONS IN THRUST, KOKKOS, OPENMP AND STDPAR.

Component	Thrust	Kokkos	OpenMP	stdpar
Multi-dimensional array	mdspan	View	in-house View	mdspan
Data allocator	thrust::device_vector	View	Unstructured mapping	vector
Loop	thrust::for_each	parallel_for	Directives	for_each_n
Reduction	thrust::transform_reduce	parallel_reduce	Directives	transform_reduce

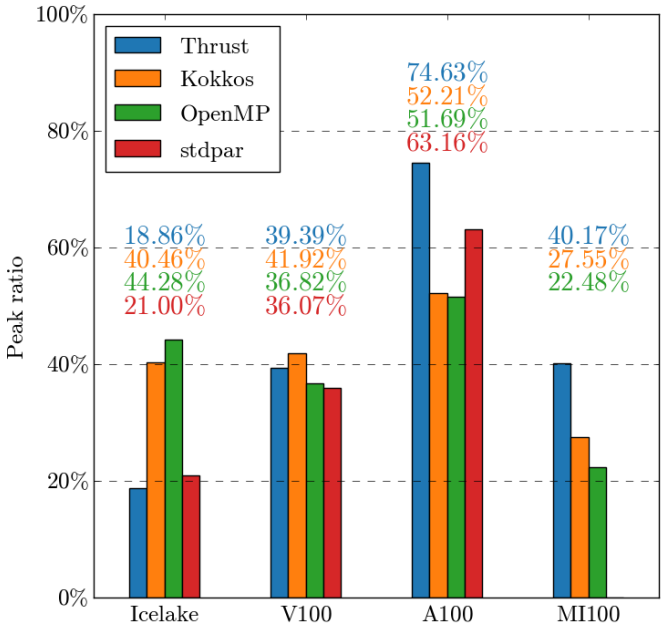

Fig. 2. Performance comparison for our 3D heat solver with Thrust, Kokkos, OpenMP, and stdpar. Peak ratio is computed by the achieved memory bandwidth evaluated with Eq. (1) divided by the hardware peak memory bandwidth shown in Table I.

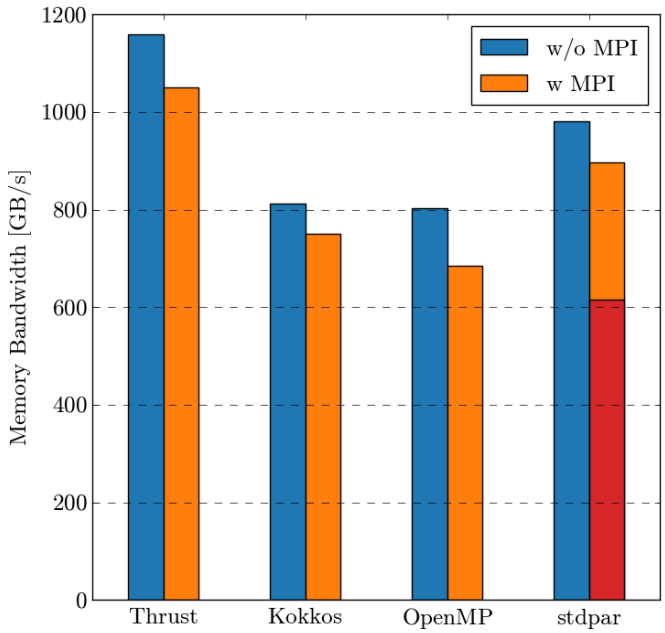

Fig. 3. Performance comparison for the 3D heat solver on A100 with and without MPI. 2 MPI processes on a single node are used. The red bar in stdpar represents the obtained performance without setting the environmental variable UCX_RNDV_FRAG_MEM_TYPE=cuda.

performance overheads from indices operations in mdspan version.

Secondly, we compare the performance of our 3D heat solver over Icelake, V100, A100, and MI100. Figure 2 shows the memory bandwidth ratio of the peak in our 3D heat solver. The achieved memory bandwidth is computed with the following equation

$$\text{Bandwidth} = N_{\text{iter}} \times N'_x \times N'_y \times N'_z \times 16/t, \qquad (1)$$

where N_{iter} is the number of iterations, $N'_x \times N'_y \times N'_z$ is the problem size at each MPI process, and t is the total elapsed time in seconds. The number 16 represents the load/store operations of double precision data in bytes, assuming a perfect and unlimited cache in each time step, but no reuse over time iterations. In this definition, the MPI communication, swapping, packing, and unpacking costs are regarded as overheads.

B. Multi processor performance of the 3D heat solver

As we have explained in subsection II-B, data is allocated on managed memory buffers in stdpar. MPI communication of the data in managed memory stages it to host memory by default, irrespective of the location where it is actually held. This behaviour results in a serious performance penalty and thus MPI + stdpar code does not perform well as described in the article [29]. With NVIDIA HPC SDK 22.5+ (or more accurately the combination of UCX 1.13.0 and OpenMPI 4.1.4), we can force MPI communications to bypass host memory with the environmental variable UCX_RNDV_FRAG_MEM_TYPE=cuda, which is critical for the performance. Figure 3 shows the performance of the 3D heat solver with and without MPI. The problem size is fixed as $(N_x, N_y, N_z) = (512, 512, 512)$ and the MPI domain decomposition is made along z direction. Other than z direction, the halo regions are exchanged by swapping. Except for stdpar, the performance degradation from w/o MPI to MPI is small due to the fast device to device communications by Nvlink. By forcing the device to device communication, the performance of stdpar version is improved by a factor of 1.4 (from red to orange bar). This feature is essential to get a good performance with MPI + stdpar implementation without language specific implementations.

IV. Cross-platform performance evaluation

Based on the implementations established in sections II and III, we evaluate and compare the performance of our kinetic plasma mini-applications with multiple implementations. We firstly evaluate a single process performance using the non-MPI version of vlp4d code [13]. Then, we investigate the performance of the MPI version of vlp4d [14]. Finally, we evaluate the P3 aspect of `stdpar` through the comparison with other programming models.

A. Performance portability metric

In the present work, we employ the performance portability metric [18], [19] defined as

$$\mathcal{P}(a,p,H) = \begin{cases} \dfrac{|H|}{\sum_{i\in H} \dfrac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise,} \end{cases} \quad (2)$$

$$e_i(a,p) = \frac{P_{a,p,i}}{R_{a,i}} \times 100\%. \quad (3)$$

Here, $e_i(a,p)$ denotes the architectural efficiency of an application a on the device i with simulation settings p based on the roofline model [30]. $P_{a,p,i}$ is the achieved GFlops of the kernel a on the device i based on the hand count metrics of kernels. The attainable performance $R_{a,i}$ is computed by roofline model [30] as

$$R_{a,i} = \min(F_i, B_i \times f_a/b_a), \quad (4)$$

where F_i and B_i are the Peak Floating point Performance in GFlops and the Peak Memory Bandwidth in GBytes/s of the device i. f_a and b_a denote the total number of floating point operations and memory accesses per grid point of a kernel a, counted by hand. H represents a set of platforms including Icelake, V100, A100, and MI100. We exclude MI100 from H for the `stdpar` programming model, since it is not available now. A low value of $\mathcal{P}(a,p,H)$ indicates the achieved performance on each architecture is insufficient.

B. GYSELA mini-apps

GYSELA mini-apps have been developed to investigate a performance portable approach which is suitable for the GYSELA 5D code [15]. In these mini-apps, we solve the 4D (2D space and 2D velocity space) Vlasov-Poisson system [31] with and without MPI domain decomposition [13], [14]. The time evolution of the 4D distribution function f is computed self-consistently by solving the 4D Vlasov and 2D Poisson equations as

$$\frac{\partial f}{\partial t} + \mathbf{v}\cdot\nabla_{\mathbf{x}} f + E(t,\mathbf{x})\cdot\nabla_{\mathbf{v}} f = 0, \quad (5)$$

with $\mathbf{x} = (x,y)$ and $\mathbf{v} = (v_x, v_y)$. The 2D Poisson equation

$$\nabla_{\mathbf{x}}\cdot E(t,\mathbf{x}) = \rho(t,\mathbf{x}) - 1, \quad (6)$$

with the ion density $\rho(t,\mathbf{x}) = \int d\mathbf{v} f(t,\mathbf{x},\mathbf{v})$ and Electric field $E(t,\mathbf{x})$. Hereinafter, "f^n" will denote the distribution function at time $n\Delta t$, with the number of iteration n and the time step size Δt. For the sake of simplicity, all directions are handled with periodic boundary conditions.

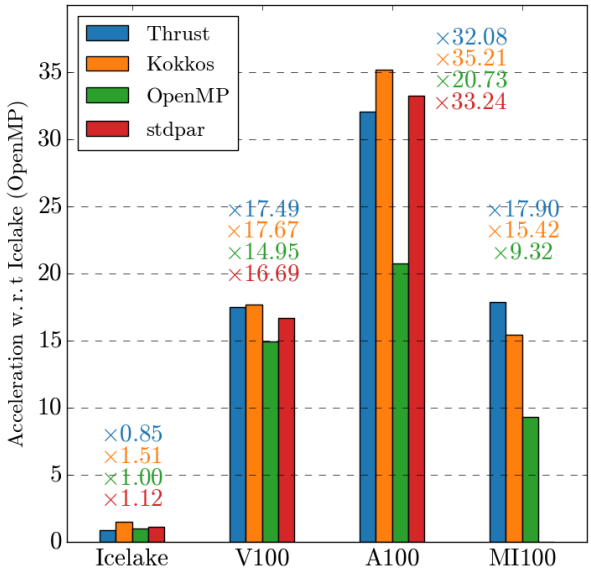

Fig. 4. Performance comparison for vlp4d with Thrust, Kokkos, OpenMP, and `stdpar`.

C. Single process performance

1) Brief description of non-MPI version of the mini-app: In the non-MPI version, we compute the time integration as shown in algorithm 2.

Algorithm 2 One time step of non-MPI version

1: **Input:** f^n, **Output:** f^{n+1}
2: 1D advection along x direction for $\Delta t/2$
3: 1D advection along y direction for $\Delta t/2$
4: Velocity space integral: Compute $\rho^{n+1/2}$
5: Field solver: Compute $E_x^{n+1/2}, E_y^{n+1/2}$
6: 1D advection along v_x direction for Δt
7: 1D advection along v_y direction for Δt
8: 1D advection along y direction for $\Delta t/2$
9: 1D advection along x direction for $\Delta t/2$

As detailed in [13], Strang's operator splitting method [32] is used to solve the Vlasov Eq. (5). The 4D advection Eq. (5) is split as a combination of 1D advection equations along x, y, v_x, and v_y directions. Each advection equation is solved using a backward semi-Lagrangian scheme with 5^{th} order Lagrange interpolation (6 points stencils). To compute the 2D density, the 4D distribution function f is integrated over velocity space. The Poisson equation is solved in Fourier space using a 2D Fast Fourier Transform (FFT) under the periodic boundary conditions in (x,y) directions. We use fftw [33], cuFFT [34], and rocFFT [35] for CPUs, NVIDIA GPUs, and AMD GPUs, respectively.

Figure 4 shows the performance of the entire mini-app in terms of acceleration with respect to the baseline

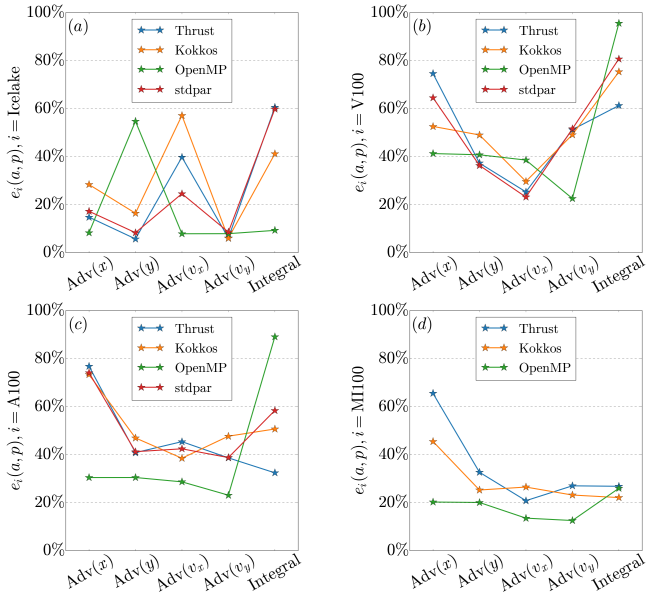

Fig. 5. Architectural efficiency $e_i\,(a,p)$ of Adv (x), Adv (y), Adv (v_x), Adv (v_y), and Integral kernels on (a) Icelake, (b) V100, (c) A100, and (d) MI100. The operational intensity f/b of each kernel is measured in average assuming a perfect and unlimited cache. The f/b of Adv (x), Adv (y), Adv (v_x), Adv (v_y), and Integral kernels are $67/16$, $67/16$, $65/16$, $65/16$, and $1/8$, respectively [13]. The ideal performance is estimated by the roofline model in Eq. (4).

OpenMP version on Icelake. The problem size p is fixed to $\left(N_x, N_y, N_{v_x}, N_{v_y}\right) = (128, 128, 128, 128)$ and the number of iterations is 128. The acceleration of GPU versions can be partially attributed to the better performance of 1D advection along x and y directions on GPUs; these kernels are called twice at every time step (see Algorithm 2). We compare the performance of advection and integral kernels in subsections IV-C2 and IV-C3.

2) Architectural efficiency of each kernel: We evaluate the architectural efficiencies $e_i\,(a,p)$ of the advection kernels in the Vlasov solver and the integral kernel in the Poisson solver. Advection kernels have high operational intensity of about 4 while Integral kernel has low operational intensity of 0.125 (close to memory bandwidth measurements). Figure 5 shows the architectural efficiencies on Icelake, V100, A100, and MI100 with each programming model.

In Kokkos, Thrust, and `stdpar` implementations, we deliberately keep a simple for loop inside functors to enhance vectorization on CPUs and allow peeling on GPUs [13]. We use `LayoutLeft` for GPUs with all programming models. For Icelake, we use `LayoutRight` with Thrust, Kokkos and `stdpar`, but use `LayoutLeft` with OpenMP. The layout abstraction capability with `Kokkos::View` or `mdspan` allows layout tuning while keeping a single codebase. This allows a good GPU performance in Kokkos, Thrust, and `stdpar` while keeping a good CPU performance. `mdspan` not only improves the readability and productivity but also performance portability where the better layout differs for CPUs and GPUs. As for the OpenMP version, the architectural efficiencies $e_i\,(a,p)$ for advection kernels are lower than other programming models on A100 and MI100.

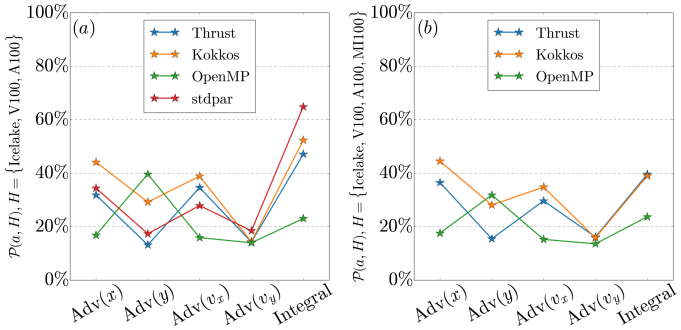

Fig. 6. Performance portability $\mathcal{P}(a,H)$ of major kernels in vlp4d. Here, a consists of "Adv (x)", "Adv (y)", "Adv (v_x)", "Adv (v_y)", and "Integral". (a) H is the set of {Icelake, V100, A100} and (b) {Icelake, V100, A100, MI100}.

In the Kokkos implementation, we use the tile size (T_x, T_y, T_z) with $(4,4,4)$ for Icelake, $(32,4,2)$ for V100 and A100, and $(64,4,2)$ for MI100. It should be noted that there is a room to optimize the tile size for a given kernel as done in [13], particularly on Icelake.

3) Performance portability of each kernel: As shown in Fig. 6 (a), the performance portability of `stdpar` version is close to that of Thrust version (2-10% difference). Given that `stdpar` relies on Thrust as backend, this is a quite reasonable consequence. The performance difference of `stdpar` and Thrust on V100 and A100 may stem from the difference in the number of assigned threads. The Kokkos version achieves the best or second best performance portability over all kernels. The `stdpar` version achieves the best performance portability for "Adv (v_y)" and "Integral" kernels. The lower performance portability with OpenMP version in Fig. 6 (b) stems from the lower performance on A100 and MI100.

D. Multi processor performance

1) Brief description of MPI version of the mini-app: To move forward to more complex settings as GYSELA, we upgrade the mini-app to use Spline interpolations for the backward semi-Lagrangian scheme, and add the MPI domain decomposition. Algorithm 3 shows the time integration scheme in the MPI version of mini-app [14]. We use the MPI domain decomposition based on the Unbalanced Recursive Bisection (URB) algorithm [36]. This version involves P2P and all_reduce MPI communications, which are performed respectively with CUDA-Aware-MPI or ROCm-Aware-MPI [37] on NVIDIA or AMD GPUs. As discussed in subsection III-B, we need to enforce device to device MPI communications for `stdpar` on GPUs to get a reasonable performance.

Contrary to the non-MPI version, we solve the 4D (x, y, v_x, v_y) advection without operator splitting. This kernel is extremely compute intense with indirect memory accesses. The use of Spline interpolations instead of Lagrange interpolation require a matrix LU decomposition that leads to read/write dependencies.

In the following measurements, we have applied the optimizations in [14] suitable for each programming model, such as vectorization, cache tuning, layout tuning, and execution policy tuning. We use `LayoutLeft` for all the devices.

Algorithm 3 One time step of MPI version

1: **Input:** f^n, **Output:** f^{n+1}
2: Halo exchange on f^n (P2P communications)
3: Compute spline coefficients along (x, y) directions: $f^n \rightarrow \eta_{\alpha,\beta}$
4: 2D advection along (x, y) directions for $\Delta t/2$
5: Velocity space integral: Compute $\rho^{n+1/2}$ (MPI_all_reduce communication)
6: Field solver: Compute $E_x^{n+1/2}, E_y^{n+1/2}$
7: Compute spline coefficients along (v_x, v_y) directions: $\eta_{\alpha,\beta} \rightarrow \eta_{\alpha,\beta,\gamma,\delta}$ (computed from $\eta_{\alpha,\beta}$ as a tensor product)
8: 4D advection along x, y, v_x, v_y directions for Δt

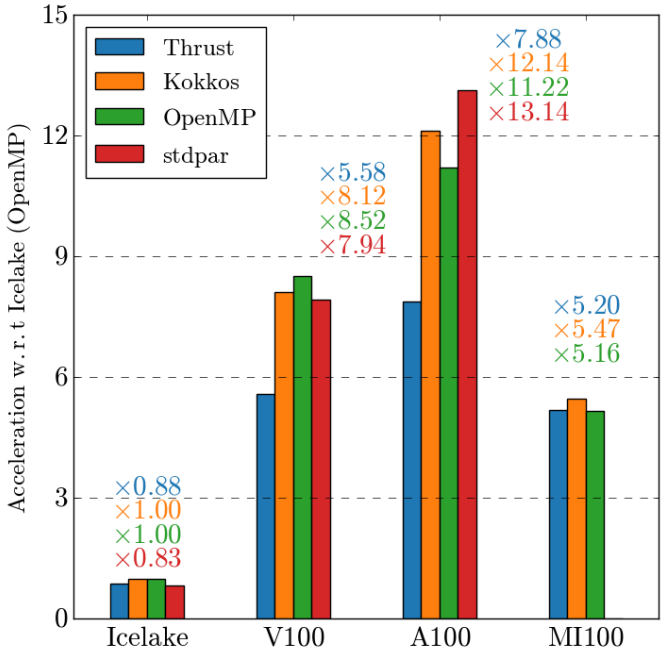

Fig. 7. Performance comparison for MPI version of vlp4d with Thrust, Kokkos, OpenMP, and `stdpar`.

Figure 7 shows the performance of the entire mini-app in terms of acceleration with respect to the baseline OpenMP version on Icelake. The problem size p is fixed to $(N_x, N_y, N_{v_x}, N_{v_y}) = (128, 128, 128, 128)$ and the number of iterations is 40. We use 2 MPI processes corresponding to 2 Icelake sockets, V100, A100, and MI100 GPUs. On Icelake, the better performance is obtained in Kokkos and OpenMP versions with the tiling capability and auto-vectorization, respectively. Performance with `stdpar` and Thrust are worse than with OpenMP version due to the lack of vectorization. The `stdpar` version exhibits performance in the range of $\pm 20\%$ to the Kokkos version on Icelake, V100, and A100. We compare the performance of costly kernels in subsections IV-D2 and IV-D3.

2) Architectural efficiency of each kernel: We evaluate the performance of "Adv2D", "Adv4D", "Spline", and "Integral" kernels. Figure 8 shows the architectural efficiencies on Icelake, V100, A100, and MI100 with each programming model.

Due to the lack of parallelism, the architectural efficiency

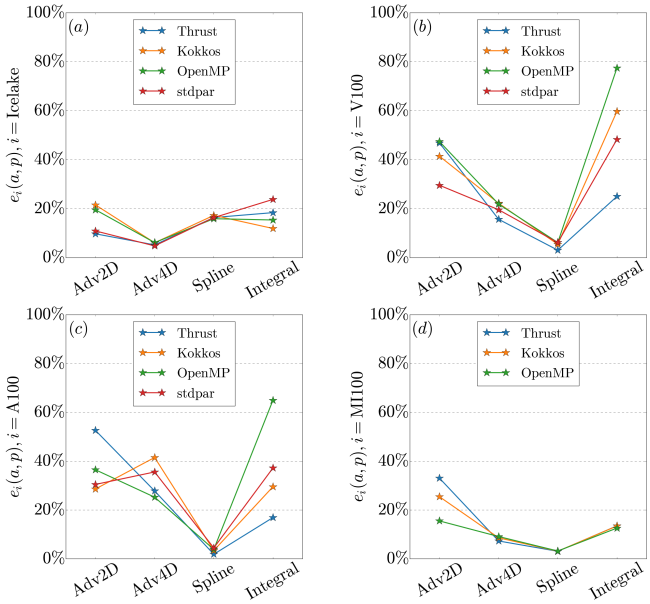

Fig. 8. Architectural efficiency $e_i\,(a, p)$ of "Adv2D", "Adv4D", "Spline" and "Integral" kernels on (a) Icelake, (b) V100, (c) A100, and (d) MI100. The operational intensity f/b of each kernel is measured in average assuming a perfect and unlimited cache. The f/b of Adv2D, Adv4D, Spline, and Integral is 86/32, 876/32, 16/16, and 1/8, respectively [14]. The ideal performance is estimated by the roofline model in Eq. (4).

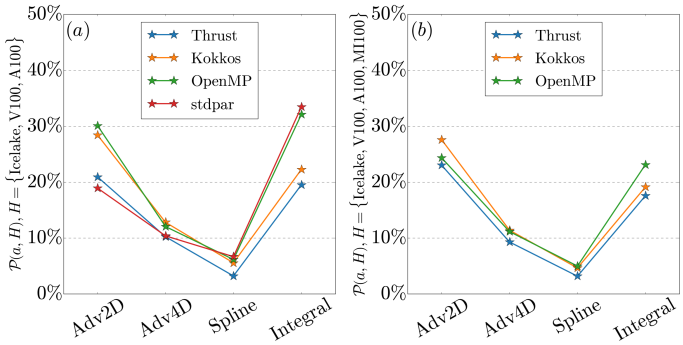

Fig. 9. Performance portability $\mathcal{P}(a, H)$ of major kernels in MPI version of vlp4d. Here, a consists of "Adv2D", "Adv4D", "Spline", and "Integral". (a) H is the set of {Icelake, V100, A100} and (b) {Icelake, V100, A100, MI100}.

of "Spline" kernel is fairly low for all GPUs (less than 10 %). The overall poor performance of "Adv4D" kernel on MI100 may stem from the lower instruction throughput on MI100 compared to NVIDIA GPUs. The doubled wavefront size in MI100 compared to the warp size in NVIDIA GPUs can halve the number of wavefront instructions on MI100 [38].

3) Performance portability of each kernel: Based on the architectural efficiency in Fig. 8, we evaluate the performance portability $\mathcal{P}(a, p, H)$ in Eq. (2). As shown in Fig. 8 (a), the `stdpar` version demonstrates the best performance portability for the "Spline" and "Integral" kernels. Kokkos version achieves the best performance portability for the compute intense "Adv4D" kernel, whereas the OpenMP version achieves the best performance portability for the "Adv2D" kernel.

V. Discussions about performance, portability, and productivity

Based on this case study, we discuss the performance, portability, and productivity achieved with `stdpar` and `mdspan` and compare against the more widespread Kokkos and OpenMP.

A. Performance portability

The performance portability achieved with `stdpar` on Intel CPU and NVIDIA GPUs is quite competitive with other programming models like Kokkos or OpenMP. Compiler auto-vectorization for `stdpar` on CPUs is less effective compared to the naive for loop, even if the `std::execution::par_unseq` execution policy is chosen that should allow the loop parallelization and vectorization. As for portability, `stdpar` is currently available only on NVIDIA platforms, whereas Kokkos, Thrust, and OpenMP are equally available on AMD platforms. Since `stdpar` is part of the standard C++ language, one can expect it to become available for AMD and Intel GPU platforms in the future.

B. Productivity and readability

Thanks to `mdspan`, we can access to multi-dimensional data in a readable and productive manner with layout abstraction in `stdpar`. Compared with raw pointers, the performance overhead is mostly negligible or small for most of programming models. With additions in C++, we can expect to rely on cartesian_product to ease the parallelization for multi-dimensional loops. Due to the lack of explicit memory space, our `stdpar` versions rely on managed memory which can cause performance issue, for example in MPI communications. With NVIDIA HPC SDK 22.5 or later, we can enforce the device to device MPI communications with the environmental variable, and thus no longer need to allocate the GPU memory with `cudaMalloc` or `thrust::device_vector` for MPI [29].

In our experience, the productivity and programmability of a framework are highly influenced by the existence of publicly available documentations and examples. As is often the case for C++ template-based libraries including Thrust, Kokkos and `stdpar`, a single mistake results in a tremendous amount of compile errors. Kokkos is well documented on GitHub [39], which eases to resolve errors and/or integrate desired Kokkos features in our codes. In that sense, implementing a code with Kokkos is easier than with Thrust and `stdpar`. In addition, it has taken a certain amount of time to explore the flexible way to implement parallel iterations and reductions with Thrust and `stdpar`. Once it is established, we can reuse the Kokkos functors for parallel operations in Thrust and `stdpar`, resulting in the significant mitigation of the programming costs in these frameworks.

OpenMP is a reasonable choice to port a large Fortran 90 code like GYSELA [15]. This is the only framework which allows to keep using the Fortran 90. Since the cost to rewrite the Fortran 90 code into C++ with a performance portable layer is significant, this is the major benefit to employ OpenMP. Though the productivity of OpenMP is a promising aspect, we have encountered two major difficulties in this case study: defining a data structure and interfacing to GPU libraries. When introducing a new data structure, we always have to carefully copy the instance of the data structure to GPUs with shallow and deep copies. Most of GPU libraries are written in an architecture specific language such as CUDA and HIP. On MI100, we need to link HIP APIs to use rocFFT and rocBlas in addition to libraries themselves. Writing an appropriate CMake file for MI100 with OpenMP is more complex than those for Thrust, Kokkos and HIP.

C. Benefit of using `stdpar`

One of the most important benefits of `stdpar` is maintainability or sustainability of a code. Since a `stdpar` code just relies on the standard C++ language (no platform specific extensions), it is expected to run on future HPC platforms. The code can also benefit from the future developments of C++ such as cartesian_product [27], linear algebra [40], and executors [41].

VI. Summary

In this paper, we evaluated the performance, portability, and productivity (P3) of kinetic plasma mini-applications with C++ parallel algorithm (`stdpar`). As well as the language standard parallelization, we also focused on the language standard high dimensional array support `mdspan`. We compared P3 aspects of `stdpar` with other performance portable programming models such as Kokkos and OpenMP.

Firstly, we explored the efficient and readable implementation with `stdpar` and `mdspan` with a 3D heat equation solver. Through the comparison with the Kokkos version, we demonstrated that the `stdpar` version can be implemented in a readable manner with abstractions of parallel loops and data structures. In `stdpar`, parallel operations can be defined by `std::for_each_n` and `std::transform_reduce` similar to `Kokkos::parallel_for` and `Kokkos::parallel_reduce` in Kokkos. Due to the lack of multi-dimensional loop support in `stdpar`, we need to map the 1D index into the multi-dimensional indices by hand. For the data structure, we can implement a Kokkos View like multi-dimensional array by coupling `mdspan` and `std::vector`. By enforcing GPU-aware device to device MPI communications, we obtained quite competitive performance with MPI + `stdpar` using standard C++ and proposed standard extensions only.

Secondly, we evaluated the performance portability metric on a kinetic plasma mini-app with and without MPI parallelization, which encapsulate key features of kinetic plasma turbulence code GYSELA. The non-MPI version solves the 4D Vlasov equation with Lagrange interpolations. The layout abstraction in Kokkos View and `mdspan` allows the suitable data layout for GPUs and CPUs while keeping a single codebase. The `stdpar` version achieves the best performance portability for "Adv (v_y)" and "Integral" kernels, whereas the

Kokkos version achieves the best or second best performance portability on all kernels. In the MPI version, we use the Spline interpolation for backward semi-Lagrangian scheme and add MPI domain decomposition. The Kokkos version achieved the best performance portability for the "Adv4D" kernel whereas the `stdpar` version demonstrated the best performance portability for the "Spline" and "Integral" kernels. The `stdpar` version showed the best overall performance on A100. Given the good overall performance of `stdpar` (in the range of $\pm 20\%$ with respect to the Kokkos version) on Icelake, V100 and A100, `stdpar` has competitive performance portability.

Finally, we discussed the performance, portability and productivity achieved with `stdpar` and `mdspan` based on this case study. Through the performance comparison of kinetic plasma mini-apps in multiple implementations, we found that `stdpar` is quite competitive in performance portability. The most important benefit to employ `stdpar` is the maintainability or sustainability of a code. Thus, `stdpar` can be a reasonable choice as an Exascale performance portable implementation assuming that it becomes available on AMD and/or Intel GPUs in the future.

ACKNOWLEDGMENT

This work was carried out using HPE SGI8600 at JAEA and FUJITSU PRIMERGY GX2570 (Wisteria/BDEC-01) at The University of Tokyo. This work was partly supported by JHPCN projects jh210049-MDH and jh220031. The author Y.A thank Dr. H. Shiba and Dr. T. Shimokawabe for the fruitful discussions about AMD GPU usage. The author Y.A also thank the computing resources for AMD GPUs provided by Information Technology Center, The University of Tokyo.

REFERENCES

[1] top500, "TOP 500," https://top500.org, Last accessed on 2022-08-08.

[2] OpenMP, "Openmp application programming interface," november 2015, https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[3] OpenACC, "Openacc 2.7 api reference card," november 2018, https://www.openacc.org/sites/default/files/inline-files/API%20Guide%202.7.pdf.

[4] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[5] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming Model Extensions for the Exascale Era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[6] Hornung, Richard D. and Keasler, Jeffrey A., "The RAJA Portability Layer: Overview and Status," Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), Tech. Rep.

[7] R. Reyes, G. Brown, R. Burns, and M. Wong, "Sycl 2020: More than meets the eye," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3388333.3388649

[8] P. Grete, F. W. Glines, and B. W. O'Shea, "K-athena: a performance portable structured grid finite volume magnetohydrodynamics code," *CoRR*, vol. abs/1905.04341, pp. 211 – 231, 2019. [Online]. Available: http://arxiv.org/abs/1905.04341

[9] D. Sunderland, B. Peterson, J. Schmidt, A. Humphrey, J. Thornock, and M. Berzins, "An overview of performance portability in the uintah runtime system through the use of kokkos," in *2016 Second International Workshop on Extreme Scale Programming Models and Middlewar (ESPM2)*, New York, NY, USA, Nov 2016, pp. 44–47. [Online]. Available: http://doi.acm.org/10.1109/ESPM2.2016.012

[10] V. Artigues, K. Kormann, M. Rampp, and K. Reuter, "Evaluation of performance portability frameworks for the implementation of a particle-in-cell code," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 11, p. e5640, 2020, e5640 cpe.5640. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5640

[11] T. R. Law, R. Kevis, S. Powell, J. Dickson, S. Maheswaran, J. A. Herdman, and S. A. Jarvis, "Performance portability of an unstructured hydrodynamics mini-application," in *Proceedings of 2018 International Workshop on Performance, Portability, and Productivity in HPC (P3HPC)*. New York, NY, USA: ACM, 2018.

[12] J. Latt, C. Coreixas, and J. Beny, "Cross-platform programming model for many-core lattice Boltzmann simulations," *PLOS ONE*, vol. 16, no. 4, pp. 1–29, 04 2021. [Online]. Available: https://doi.org/10.1371/journal.pone.0250306

[13] Y. Asahi, G. Latu, V. Grandgirard, and J. Bigot, "Performance portable implementation of a kinetic plasma simulation mini-app," in *Accelerator Programming Using Directives*, ser. series, S. Wienke and S. Bhalachandra, Eds. Cham: Springer International Publishing, 2020, pp. 117–139.

[14] Y. Asahi, G. Latu, J. Bigot, and V. Grandgirard, "Optimization strategy for a performance portable vlasov code," in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2021, pp. 79–91.

[15] V. Grandgirard, J. Abiteboul, J. Bigot, T. Cartier-Michaud, N. Crouseilles, G. Dif-Pradalier *et al.*, "A 5D gyrokinetic full-f global semi-Lagrangian code for flux-driven ion turbulence simulations," *Computer Physics Communications*, vol. 207, pp. 35 – 68, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465516301230

[16] D. S. Hollman, B. A. Lelbach, H. C. Edwards, M. Hoemmen, D. Sunderland, and C. R. Trott, "mdspan in c++: A case study in the integration of performance portable features into international language standards," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 60–70.

[17] NVIDIA, "Thrust quick start guide," May 2022, https://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf, Last accessed on 2022-07-12.

[18] S. Pennycook, J. Sewall, and V. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, vol. 92, pp. 947 – 958, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17300559

[19] S. J. Pennycook and J. D. Sewall, "Revisiting a metric for performance portability," in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2021, pp. 1–9.

[20] *Reference implementation of mdspan targeting C++23*, "https://github.com/kokkos/mdspan", Last accessed on 2022-08-08.

[21] *P3-miniapps*, "https://github.com/yasahi-hpc/P3-miniapps", Last accessed on 2022-08-08.

[22] NVIDIA, "NVIDIA Tesla V100," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[23] NVIDIA, "NVIDIA A100 Tensor Core GPU Architecture," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2018.

[24] AMD, "AMD CDNA ARCHITECTURE," https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf, 2020.

[25] Intel, "IntelXeon Platinum 8360Y Processor (54 M Cache, 2.40 GHz)," https://www.intel.com/content/www/us/en/products/sku/212459/intel-xeon-platinum-8360y-processor-54m-cache-2-40-ghz/specifications.html.

[26] C. Daley, H. Ahmed, S. Williams, and N. Wright, "A case study of porting hpgmg from cuda to openmp target offload," in *OpenMP: Portable Multi-Level Parallelism on Modern Systems*, K. Milfeld, B. R. de Supinski, L. Koesterke, and J. Klinkenberg, Eds. Cham: Springer International Publishing, 2020, pp. 37–51.

[27] S. Brand and M. Dominiak, "P2374R3 views::cartesian_product," 12 2021, https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2374r3.html, Last accessed on 2022-06-30.

[28] Advanced Micro Devices, "rocThrust Documentation," June 2022, https://rocthrust.readthedocs.io/_/downloads/en/latest/pdf/, Last accessed on 2022-07-12.

[29] G. B. Jonas Latt, Christophe Guy Coreixas and J. Larkin, "Multi-gpu programming with standard parallel c++, part 2," April 2022, https://developer.nvidia.com/blog/multi-gpu-programming-with-standard-parallel-c-part-2/, Last accessed on 2022-06-30.

[30] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[31] N. Crouseilles, G. Latu, and E. Sonnendrücker, "A parallel vlasov solver based on local cubic spline interpolation on patches," *Journal of Computational Physics*, vol. 228, no. 5, pp. 1429 – 1446, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999108005652

[32] G. Strang, "On the Construction and Comparison of Difference Schemes," *SIAM Journal on Numerical Analysis*, vol. 5, no. 3, pp. 506–517, 1968. [Online]. Available: https://doi.org/10.1137/0705041

[33] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[34] NVIDIA, "cufft library user's guide," April 2021, https://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf, Last accessed on 2021-05-06.

[35] Advanced Micro Devices, "rocFFT Documentation," June 2022, https://rocfft.readthedocs.io/en/rocm-5.2.0/, Last accessed on 2022-07-05.

[36] M. T. Jones and P. E. Plassmann, "Computational results for parallel unstructured mesh computations," *Computing Systems in Engineering*, vol. 5, no. 4, pp. 297 – 309, 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0956052194900132

[37] K. Shafie Khorassani, J. Hashmi, C.-H. Chu, C.-C. Chen, H. Subramoni, and D. K. Panda, "Designing a ROCm-Aware MPI Library for AMD GPUs: Early Experiences," in *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24 – July 2, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 118–136. [Online]. Available: https://doi.org/10.1007/978-3-030-78713-4_7

[38] M. Haseeb, N. Ding, J. Deslippe, and M. Awan, "Evaluating Performance and Portability of a core bioinformatics kernel on multiple vendor GPUs," in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2021, pp. 68–78.

[39] *Kokkos C++ Performance Portability Programming EcoSystem*, "https://github.com/kokkos/kokkos", Last accessed on 2021-04-01.

[40] M. Hoemmen, D. Hollman, C. Trott, D. Sunderland, N. Liber, L.-T. Lo, D. Lebrun-Grandie, G. Lopez, P. Caday, S. Knepper, P. Luszczek, and T. Costa, "P1673r3: A free function linear algebra interface based on the blas," standard P1673R3, 2021-04 2021. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1673r3.pdf

[41] J. Hoberock, M. Garland, C. M. Chris Kohlhoff, H. C. Edwards, G. Brown, and D. S. Hollman, "A Unified Executors Proposal for C++," 9 2020, https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0443r14.html, Last accessed on 2022-06-30.

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: [PERFORMANCE PORTABLE VLASOV CODE WITH C++ PARALLEL ALGORITHM]

A. Abstract

This appendix is the artifact description of the paper entitled "Performance portable Vlasov code with C++ parallel algorithm". It includes the basic instruction for installation of "P3-miniapps" and experiment workflows. The expected results and the data evaluation method in the paper are also demonstrated.

B. Description

1) Check-list (artifact meta information):

- **Algorithm: Semi-Lagrangian method, Spline interpolation, Unbalanced Recursive Bisection algorithm**
- **Program:** `heat3d_mpi`**,** `vlp4d`**, and** `vlp4d_mpi` **[https://github.com/yasahi-hpc/P3-miniapps]**
- **Compilation: Instructions on how to build the mini-app P3-miniapps are available on the GitHub page.**
- **Run-time environment: See the GitHub page.**
- **Hardware: Intel Xeon Platinum 8360Y (Icelake), AMD MI100, NVIDIA V100 (PCIE 32GB), and A100 (PCIE 40GB).**
- **Execution: See "Experiment workflow" section.**
- **Output: See "Evaluation and expected result" section.**
- **Experiment workflow: See "Experiment workflow" section.**
- **Publicly available?: Yes**

2) How software can be obtained (if available): The source code `heat3d`, `heat3d_mpi`, `vlp4d`, and `vlp4d_mpi` are publicly available on the GitHub page https://github.com/yasahi-hpc/P3-miniapps. (under `miniapps` directory)

3) Hardware dependencies: We have tested on Intel Xeon Platinum 8360Y (Icelake), AMD MI100 GPU, NVIDIA V100, and A100 GPUs.

TABLE I
BUILD COMMANDS FOR EACH DEVICE

DEVICE	programming_model	compiler_name	backend
Icelake	THRUST	g++	OPENMP
	KOKKOS	icpc	
	OPENMP	icpc	
	STDPAR	nvc++	
P100 V100 A100	CUDA	g++	CUDA
	THRUST	g++	
	KOKKOS	nvcc_wrapper	
	OPENMP	nvc++	
	STDPAR	nvc++	
MI100	HIP	hipcc	HIP
	THRUST	hipcc	
	KOKKOS	hipcc	
	OPENMP	hipcc	

4) Software dependencies: This software relies on external libraries including Kokkos [https://github.com/kokkos/kokkos], `mdspan` [https://github.com/kokkos/mdspan], and fftw [http://www.fftw.org]. Kokkos and `mdspan` are included as submodules. CUDA-Aware-MPI or ROCm-Aware-MPI are needed for NVIDIA or AMD GPUs.

C. Installation

We have provided 4 mini-applications `heat3d`, `heat3d_mpi`, `vlp4d`, and `vlp4d_mpi`. Each application is parallelized with "(MPI+) X" programming model, where "X" includes `stdpar` (C++ parallel algorithm), OpenMP, OpenACC, Kokkos, Thrust, CUDA and HIP. We rely on CMake to build the applications. One may compile with the following CMake command. Table I shows the available combinations of `<programming_model>`, `<compiler_name>` and `<backend>` for each `<DEVICE>`. `<app_name>` should be chosen from the mini-application names listed above. The compilers and compile options for each version are listed in Table II. The detailed instruction is found at the GitHub page (Compile section).

```
cmake \
-DCMAKE_CXX_COMPILER=<compiler_name> \
-DBUILD_TESTING=OFF \
-DCMAKE_BUILD_TYPE=<build_type> \
-DPROGRAMMING_MODEL=<programming_model>\
-DBACKEND=<backend> \
-DAPPLICATION=<app_name>
```

D. Experiment workflow

Job scripts used in the benchmark are stored in *wk* directory each of which is named as

```
sub_<programming_model>_<app_name>_<DEVICE>.sh
```

1) Basic benchmarks with heat3d_mpi (Section III): For all the measurements in section III, we used `heat3d_mpi`.

1) Compilation
   We firstly build as follows (A100 with `stdpar`):

```
cmake -B build \
-DCMAKE_CXX_COMPILER=nvc++ \
-DBUILD_TESTING=OFF \
-DCMAKE_BUILD_TYPE=Release \
-DPROGRAMMING_MODEL=STDPAR \
-DBACKEND=CUDA \
-DAPPLICATION=heat3d_mpi
cmake --build build
```

   Default version relies on `mdspan`. To use raw pointers, we further need to add following options.

```
-DACCESS_VIA_RAW_POINTERS=ON \
-DRANGE_POLICY_1D=ON
```

2) Run
   For the performance comparison with and without MPI, we used the following commands. See Run section in `docs/heat3d.md` on GitHub for detail.

```
# 1 MPI
./heat3d_mpi --px 1 --py 1 --pz 1 \
--nx 512 --ny 512 --nz 512 \
--nbiter 1000 --freq_diag 0

# 2 MPI
export UCX_RNDV_FRAG_MEM_TYPE=cuda
./heat3d_mpi --px 1 --py 1 --pz 2 \
--nx 512 --ny 512 --nz 256 \
--nbiter 1000 --freq_diag 0
```

TABLE II
COMPILER AND COMPILER FLAGS USED FOR EACH VERSION

Version	Compiler	Compiler flags
Icelake (Thrust)	CUDA/11.2.152	-O3 –generate-code=arch=compute_80,code=[compute_80,sm_80] -fopenmp -std=c++17 -DTHRUST_DEVICE_SYSTEM=THRUST_DEVICE_SYSTEM_OMP
Icelake (Kokkos)	ICC 2021.2.0	-O3 -xCORE-AVX512 -qopenmp -std=gnu++17
Icelake (OpenMP)	ICC 2021.2.0	-O3 -ipo -xHOST -qopenmp -std=gnu++17
Icelake (stdpar)	NVIDIA HPC SDK 22.5	-O3 -stdpar=multicore -mp -fast –c++17 –gnu_extensions
V100 (Thrust/CUDA)	CUDA/11.4.100	-O3 –generate-code=arch=compute_70,code=[compute_70,sm_70] -std=c++17
V100 (Kokkos)	CUDA/11.4.100	-O3 -mrtm -arch=sm_70 -Xcompiler -fopenmp -std=c++17
V100 (OpenMP)	NVIDIA HPC SDK 22.3	-O3 -Minfo=accel -mp=gpu -mcmodel=medium -fast –c++17 –gnu_extensions
V100 (stdpar)	NVIDIA HPC SDK 22.3	-O3 -stdpar=gpu -fast –c++17 –gnu_extensions
A100 (Thrust/CUDA)	CUDA/11.2.152	-O3 –generate-code=arch=compute_80,code=[compute_80,sm_80] -std=c++17
A100 (Kokkos)	CUDA/11.2.152	-O3 -mrtm -arch=sm_80 -Xcompiler -fopenmp -std=c++17
A100 (OpenMP)	NVIDIA HPC SDK 22.5	-O3 -Minfo=accel -mp=gpu -mcmodel=medium -fast –c++17 –gnu_extensions
A100 (stdpar)	NVIDIA HPC SDK 22.5	-O3 -stdpar=gpu -fast –c++17 –gnu_extensions
MI100 (Thrust/HIP)	rocm 5.2.21151	-O3 -x hip –offload-arch=gfx908 -std=gnu++17
MI100 (Kokkos)	rocm 5.2.21151	-O3 -fopenmp -fno-gpu-rdc –amdgpu-target=gfx908 -std=gnu++17
MI100 (OpenMP)	rocm 5.2.21151	-O3 -fopenmp=libomp -fopenmp-targets=amdgcn-amd-amdhsa \ -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx908 -std=gnu++17

3) Expected results
Finally, we get the following performance results in standard output file in the ascii format.

```
L2_norm: 0.00355178
3D Range policy
STDPAR backend
Elapsed time: 1.19954 [s]
Bandwidth/GPU: 895.13 [GB/s]
Flops/GPU: 503.511 [GFlops]

total 1.19954 [s], 1 calls
MainLoop 1.19951 [s], 1000 calls
Heat 0.996968 [s], 1000 calls
HaloPack 0.0648857 [s], 3000 calls
HaloUnpack 0.0494859 [s], 3000 calls
HaloComm 0.0878454 [s], 3000 calls
IO 0 [s], 0 calls
```

2) Cross-Platform performance evaluation with vlp4d and vlp4d_mpi (Section IV): For all the measurements in section IV, we used `vlp4d` and `vlp4d_mpi`.

1) Compilation
Following are the compile commands for `vlp4d` and `vlp4d_mpi` on A100 with Kokkos (assuming Kokkos is pre-installed).

```
cmake -B build \
-DCMAKE_CXX_COMPILER=nvcc_wrapper \
-DBUILD_TESTING=OFF \
-DCMAKE_BUILD_TYPE=Release \
-DPROGRAMMING_MODEL=KOKKOS \
-DBACKEND=CUDA \
-DAPPLICATION=vlp4d # or vlp4d_mpi
cmake --build build
```

For `vlp4d`, we added `-DMDRange3D=ON` option for better performance. If Kokkos is not installed, one needs to add Kokkos options for installation like

```
-DKokkos_ENABLE_CUDA=On \
-DKokkos_ENABLE_CUDA_LAMBDA=On \
-DKokkos_ARCH_AMPERE80=On
```

2) Run
See Run section in `docs/vlp4d.md` on GitHub for detail.

3) Expected results
One may find the standard output file in ascii format showing the timing at the bottom. The timings look like

```
total 4.55461 [s], 1 calls
MainLoop 4.51995 [s], 40 calls
pack 0.122992 [s], 40 calls
comm 0.037724 [s], 40 calls
unpack 0.061194 [s], 40 calls
advec2D 0.362362 [s], 40 calls
advec4D 1.36502 [s], 40 calls
field 0.148563 [s], 80 calls
all_reduce 0.13413 [s], 80 calls
Fourier 0.0268526 [s], 80 calls
diag 0.0541387 [s], 80 calls
splinecoeff_xy 0.797317 [s], 40 calls
splinecoeff_vxvy 1.25026 [s], 40 calls
```

E. Evaluation and expected result

We evaluate the performance based on the standard output file. All measurements have been conducted 10 times and the averaged values are used as performance data. Each column denotes the kernel name, total elapsed time in seconds, and the number of call counts. The elapsed time s of a given kernel for a single iteration can be computed by

$$\text{total elapsed time [s]}/\text{number of call counts}$$

The Flops and memory bandwidth of a given kernel are computed by the following formula

$$\begin{aligned} \text{Flops} &= Nf/s \\ \text{Bytes/s} &= Nb/s, \end{aligned}$$

where f and b denote the total amount of floating point operation and memory accesses per grid point. N represent the total number of grid points ans s is the elapsed time of a given kernel for a single iteration. f and b for `vlp4d` and `vlp4d_mpi` presented in Figs. 5 and 8 of the paper (section IV) are the analytical estimates from the source code.