ESAIM: PROCEEDINGS AND SURVEYS, June 2018, Vol. 63, p. 152-178
L. Grigori, C. Japhet, P. Moireau, Editors

BUILDING AND AUTO-TUNING COMPUTING KERNELS: EXPERIMENTING WITH BOAST AND STARPU IN THE GYSELA CODE [*]

Julien Bigot[1], Virginie Grandgirard[2], Guillaume Latu[2], Jean-Francois Mehaut[3], Luís Felipe Millani[3], Chantal Passeron[2], Steven Quinito Masnada[3], Jérôme Richard[4] and Brice Videau[5]

Abstract. Modeling turbulent transport is a major goal in order to predict confinement performance in a tokamak plasma. The gyrokinetic framework considers a computational domain in five dimensions to look at kinetic issues in a plasma; this leads to huge computational needs. Therefore, optimization of the code is an especially important aspect, especially since coprocessors and complex manycore architectures are foreseen as building blocks for Exascale systems. This project aims to evaluate the applicability of two auto-tuning approaches with the BOAST and StarPU tools on the GYSELA code in order to circumvent performance portability issues. A specific computation intensive kernel is considered in order to evaluate the benefit of these methods. StarPU enables to match the performance and even sometimes outperform the hand-optimized version of the code while leaving scheduling choices to an automated process. BOAST on the other hand reveals to be well suited to get a gain in terms of execution time on four architectures. Speedups in-between 1.9 and 5.7 are obtained on a cornerstone computation intensive kernel.

1. Introduction

The optimization of high-performance computing (HPC) scientific simulation codes is critical to prevent wasting resources. For example, simulations using the GYSELA [12] code use a few hundred million CPU hours every year. With the cost estimate of about four euro-cents per CPU hour (private communication of GENCI[1]); a 10% improvement in performance is worth more than 400 k.Euro every year. On top of that, the new features supported in the code require more and more computing power that only the latest super-computers can provide. This requires porting and optimizing the code for each new machine. This remained somewhat manageable during the past decades when super-computer architectures were nearly standardized with clusters of nodes with few (1-32) general-purpose x86 CPU cores. However, this period seems to have come to an end. Of course, nobody can predict what the hardware architecture used to reach Exascale will look like exactly.

[*] *This work was supported by the Energy oriented Centre of Excellence (EoCoE), grant agreement number 676629, funded within the Horizon2020 framework of the European Union and by the C2S@Exa Inria Project Lab.*

[1] Maison de la Simulation, CEA, CNRS, Univ. Paris-Sud, UVSQ, Université Paris-Saclay, 91191 Gif-sur-Yvette, France
[2] CEA, IRFM, F-13108 Saint-Paul-lez-Durance
[3] Laboratoire d'Informatique de Grenoble/Université Grenoble Alpes
[4] Univ. Lyon, Inria, CNRS, ENS de Lyon, Univ. Claude-Bernard Lyon 1, LIP
[5] Laboratoire d'Informatique de Grenoble/CNRS

[1]http://www.genci.fr/en/organization

It is however difficult not to notice the trend of increasingly heterogeneous and complex architectures in the top500 [17] list of fastest computers with GPUs and MICs for example. Achieving high-performance on these architectures is difficult. The porting process also has to be done again and again due to the distinct available architectures.

In light of these challenges, the perfect HPC code would need to expose multiple conflicting properties at once: **i)** high-performance, **ii)** portability (including portability of performance), **iii)** maintainability and readability. Auto-tuning is a promising approach that claims to solve this dilemma by improving the portability of performance with only limited impact on the maintainability and readability. The underlying idea is to rely on an automated process to make optimization choices, evaluate and revise them for the chosen target platform through a feedback loop. While applying this approach to whole applications could be cumbersome, it is well suited for computation intensive parts of the code that can be overhauled as *computation kernels*. Thus, one can focus on the optimization and enhancement of parts of the code that represent main bottlenecks. Two variants of auto-tuning are possible depending on whether the process is applied before code execution (*static* auto-tuning) or together with the code execution (*dynamic* auto-tuning). Each one has its pros and cons: more information is available during execution, however letting the tuning process compete for resources with the application can have an impact on performance.

The remaining of this paper evaluates both a dynamic and a static auto-tuning approach on the GYSELA code. In Section 2 we introduce the code and present one of its most computation intensive kernels on which we apply the auto-tuning process: a 2D advection. In Section 3 we present StarPU [1,2], a runtime that enables dynamic auto-tuning through a task-based programming model. We demonstrate how it can be used to implement the 2D advection kernel and evaluate the performance of the resulting code. In Section 4 we present BOAST [18,19], a meta-programming framework that enables static auto-tuning through an embedded domain-specific language used to describe the kernels and their possible optimization. We use this to implement a part of the 2D advection kernel and show the performance gain that can be obtained over multiple computer architectures. Section 5 concludes the paper and presents some perspectives.

2. Context

Key factors that determine the performance of magnetic plasma containment devices as potential fusion reactors are the transport of heat, particles, and momentum. To study turbulent transport and to model Tokamak fusion plasmas, several parallel codes have been designed over the years. In the last decade, the simulation of turbulent fusion plasmas in Tokamak devices has improved a lot, notably thanks to the availability of large computers. Computational resources available nowadays have allowed the development of several petascale codes based on the well-established gyrokinetic framework. In this article, we focus on the GYSELA [12] code.

2.1. The GYSELA code

The GYSELA code is a non-linear 5D global gyrokinetic full-f code which performs flux-driven simulations of ion temperature gradient driven turbulence (ITG) in the electrostatic limit with adiabatic electrons. The five dimensions correspond to three dimensions in toroidal geometry (r, θ, φ) and 2D in velocity space (parallel velocity $v_{\parallel}$ and perpendicular velocity $v_{\perp}$ to the magnetic field). It solves the standard gyrokinetic equation for the full-f distribution function, *i.e.* no assumption on scale separation between equilibrium and perturbations is done. This 5D equation is self-consistently coupled to a 3D quasineutrality equation. The code also includes other features not described here (ion-ion collisions, several kind of heat sources). The code has the originality to be based on a semi-Lagrangian scheme [16] and it is parallelized using an hybrid OpenMP/MPI paradigm [10,13]. Practically, at each time step both Vlasov and Poisson equations are solved successively.

The relative efficiency of this application (weak scaling starting from 8k cores) is higher than 95% at 64k cores on several supercomputers [5]. Producing physics results with this tool necessitates large CPU resources. Moreover, it is expected that the needs will increase in the near future (due to high resolution for modeling ITER

Tokamak and addition of kinetic electrons mainly). Thus, adapting the code to up-to-date parallel architectures is a key issue. For this purpose, it is important to understand new hardware, their advantages and limitations.

In this paper, we investigate the GYSELA Vlasov solver which represents the most costly part of the code (more than 98% of a sequential run with no diagnostics or outputs). It describes the time evolution of the distribution function of particles $\bar{f}(r, \theta, \varphi, v_\parallel, \mu)$ where μ the magnetic momentum (proportional to $v_\perp^2$) is an adiabatic invariant. So it plays the role of a parameter in Vlasov equation. The usual way to perform a single *advection* (transport mechanism caused by the flow) in the GYSELA code [12] consists of a series of directional advections: $(\hat{v_\parallel}/2, \hat{\varphi}/2, \hat{r\theta}, \hat{\varphi}/2, \hat{v_\parallel}/2)$. Each directional advection is performed with the semi-Lagrangian scheme. This Strang-splitting converges in $O(\Delta t^2)$. It decomposes one time step into four 1D advections and one central 2D advection in the poloidal cross-section (vertical cut at a given toroidal location φ). This paper mainly focuses on the 2D advection along dimension r, θ which represents a significant fraction (10% up to 25% depending on the physical case) of the cost of the Vlasov solver. Let us define $\mathcal{X}_G = (r, \theta)$, the 2D advection equation reads:

$$B_{\parallel s}^* \partial_t \bar{f} + \overrightarrow{\nabla} \cdot \left(B_{\parallel s}^* \frac{d\mathcal{X}_G}{dt} \bar{f} \right) = 0 \quad (\hat{\mathcal{X}_G} \text{ operator}). \tag{1}$$

$B_{\parallel s}^*$ being the scalar representing the volume element in the guiding-center velocity space (further details can be found in [12]).

2.2. The 2D advection kernel

```
   Input  : $\bar{f}^\star(r, \theta, \varphi, v_\parallel, \mu)$, $Electric\_field$
   Output : $\bar{f}^\diamond(r, \theta, \varphi, v_\parallel, \mu)$

1  for $\mu_m$ do in parallel // MPI
2      for $v_{\parallel l}$ do in parallel // MPI
3          for $\varphi_k$ do in parallel // MPI+OpenMP
4              /* Substep 1: Compute velocities                                    */
5              $Velocity\_field \leftarrow derived\ from(Electric\_field, \Delta t)$;
6              /* Substep 2: Prepare interpolation                                 */
7              compute spline coefficients if needed;
8              forall $r_i, \theta_j$ do // Substep 3: Get displacement field
9                  $[\Delta r_{(i,j)}, \Delta\theta_{(i,j)}] \leftarrow Compute\_displacement(Velocity\_field, k, l, m, \Delta t)$;
10             forall $r_i, \theta_j$ do // Substep 4: Compute interpolations
11                 $\bar{f}^\diamond(r_i, \theta_j, \varphi_k, v_{\parallel l}, \mu_m) = interpolate(\bar{f}^\star(r_i - \Delta r_{(i,j)}, \theta_j - \Delta\theta_{(i,j)}, \varphi_k, v_{\parallel l}, \mu_m))$;
12
13
14
15
```

Algorithm 1: Advection along r, θ dimensions on the $\bar{f}^\star$ distribution function

The 2D advection kernel along r, θ defined in Eq. 1 is solved at each time step within the GYSELA code. The choice of a good interpolation method in a semi-Lagrangian scheme is crucial. It determines the numerical quality of the scheme and its computational cost. The tensor product is classically employed to achieve multi-dimensional interpolations combining adequately one-dimensional interpolations. The cubic spline scheme is known to be a well-balanced compromise between cost and quality for plasma simulations. It was the only scheme used in the GYSELA code in production until recently. Nevertheless, cubic splines lead to a rather complex computational kernel because spline coefficients have to be derived as an extra step through linear system solving. We have observed that high order Lagrange polynomials (order 5 or 7) can be competitive in terms of accuracy and performance. The Lagrange interpolator is simpler and thus gives us the opportunity to easily investigate lots of optimization alternatives. We therefore apply the StarPU approach (Section 3) on both the cubic spline and Lagrange interpolations, but we focus the BOAST study (Section 4) on the Lagrange version only. Algorithm 1 shows the main steps composing the 2D advection kernel. The parallel domain decomposition is over dimensions $\mu, \varphi, v_{\parallel}$, therefore the poloidal plane ($r = *, \theta = *$) is known entirely in the local MPI process for a given $(\mu_m, {v_{\parallel}}_l, \varphi_k)$ value.

The MPI and OpenMP parallelizations occur over outer loops $\mu, \varphi, v_{\parallel}$. Within this parallel loop nest, one can distinguish four substeps: 1) derive velocities, 2) prepare the interpolation, 3) compute the displacement field (particles move), and 4) effective interpolations. The following two sections focus on the optimization of these substeps on a single-node subdomain as the external MPI parallel loops simply consist in applying the same computation on each subdomain. Section 3 covers the four substeps and their intra-node parallelization replacing OpenMP by StarPU. Section 4 focuses on the last two substeps using BOAST. This focus is motivated by the fact that these two substeps represent the largest part of computational cost of Algorithm 1. Also, the first substeps do not fit well into the BOAST framework for two reasons: i) the number of code lines is quite large there and it is more desirable to focus on short kernels (a very few hundreds of lines), ii) this part is mainly memory bound and executing it apart from the main code will not be representative of what is going on within a GYSELA run.

3. Task-based optimization

HPC task-based programming models such as Legion [3], OmpSs [7], OpenMP 4 [14], Parsec [20], Peppher [4], StarPU [2] or XKaapi [11] offer a promising way to reach high-performance on complex hardware as well as performance portability over a wide range of architectures. They aim to split the computational work of applications into small units of computation called tasks. Each task requires data as input and produces output data. Tasks inputs and outputs are connected so as to form a directed acyclic graph (DAG) where the nodes are tasks and the edges represent data dependencies between them. The choice of the task execution order is done at runtime by a scheduler according to their dependencies and the targeted hardware. When compared with the usual statically scheduled fork-join model [8], this approach offers two advantages. On one hand, it enables easily expressing task-parallelism in addition to data parallelism. On the other hand, it eases performance portability in the case of work imbalance between processing units either due to irregular workloads or heterogeneous hardware such as CPU+GPU systems or CPUs with frequency throttling. However, the runtime scheduling also introduces overheads that should be carefully controlled.

3.1. **The StarPU task runtime**

StarPU [1,2] is one of the runtime systems that supports the execution of task graphs. Its tasks are typed by "codelets" that can have multiple implementations with different performance characteristics or hardware targets (CPU, GPU, etc.). This enables StarPU schedulers to use a performance model to select the best implementation of each task depending on the available execution resources. StarPU also features a data management layer that handles the memory buffers manipulated by tasks. It offers a software virtual shared memory which ensures data consistency when using accelerator with distinct address space and supports data-prefetch. This layer enables application developers to explicitly define data partitioning and which data pieces may be accessed by tasks during submission. Memory handled by this layer can either be pre-allocated by the user for the whole duration of the DAG execution or allocated by StarPU ("Temporary" or "Scratch" buffers). Temporary buffers are dynamically allocated as required to transfer data between tasks while scratch buffers are allocated once per thread and can thus only be used inside tasks and not to transfer information between them.

StarPU supports multiple programming interfaces such as a pragma-based C extension [9] or OpenMP for example. For this paper, we have chosen to use the C native API (however the API is also available for FORTRAN). In this API, codelets are represented by a dedicated structure. Their implementations are specified by functions whose pointers are stored in the structure. Tasks are submitted to the scheduler by calling functions of the API that require the codelet specifying the task type and the memory buffers to use as input and output of the task. The access mode of the buffers (IN, OUT or INOUT) and the order of submission is used to automatically deduce data dependencies using the usual data hazard model. This abstraction is known as the "sequential data consistency". The typical usage is to submit all tasks from a single "master" thread while the others are dedicated to executing the tasks. Once the graph has been submitted to the scheduler, it is executed by a set of workers according to the dependencies (eg. tasks with no dependency are executed first, enabling further tasks to be executed and so on until there is nothing left). Finally, another call synchronizes the master thread with the workers threads, waiting for the execution of all tasks of the scheduled graph to finish.

3.2. The gysela 2D advection dataflow

In order to use a task-based runtime such as StarPU to execute the 2D advection in gysela, a preliminary step is to identify the dataflow of the algorithm, *i.e.* the different types of tasks and the data they take as input and produce. Once this is done, a second step is to map these to actual StarPU tasks and memory buffers. This is typically done by grouping logical tasks together and reusing memory buffers for distinct data. The dataflow for the spline variant of the 2D advection that was presented in Algorithm 1 is presented in Figure 1. Bars at the bottom of Figure 1 describe the four part of Algorithm 1 (respectively the lines 5, 7, 8 and 10). It is discussed in this section while a mapping to StarPU concepts will be proposed in Section 3.3.

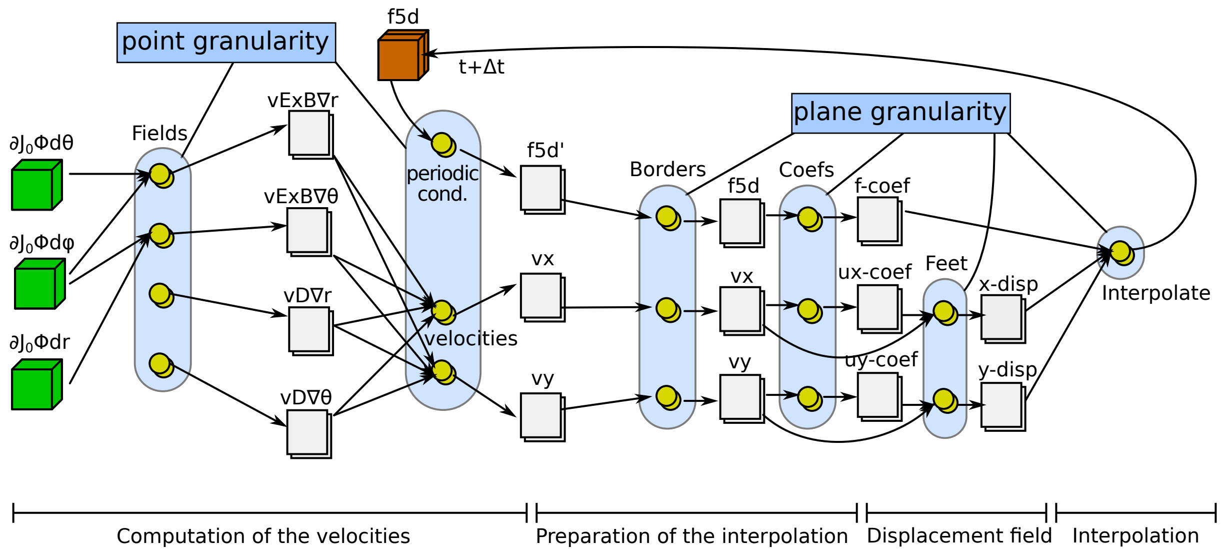

Figure 1. Dataflow for one time step of the spline-based 2D advection. Green cubes represent input buffers (read-only). The brown cube represents the input-output buffer (read-write). Grey squares represent intermediate data. Yellow circles represent bags of tasks of the same type (instances of the same codelet). Arrows are data dependencies. Blue rectangles are the finest data granularity supported by tasks (considering both input and output data).

On Figure 1, one can first identify the inputs and output of the 2D advection. In green are the derivatives of the electric field (3D in space) used to compute the displacement of particles. In brown is the particle distribution function (5D in phase space) wherein the advected values are computed in-place. The derivatives of the electric field and the particle distribution function change every time step resulting in the re-computation of the whole dataflow at each time step with different input data. One can note that some tasks at the beginning of the dataflow have no input (such as those which compute $vD\nabla r$ and $vD\nabla\theta$ data buffers). This is due to the fact that the constants that do not evolve from one time step to the other are not represented on the figure and that the only values used by these tasks are such constants. The cost of storing their result in terms of both memory and memory bandwidth has led to the choice to recompute them at each time step instead.

One can also identify the 16 different types of tasks present in the dataflow. Amongst those, some can be parallelized up to the granularity of a single point of the 5D array while others are challenging to be parallelized further than at the granularity of a (r, θ) 2D plane. The four groups of tasks entitled `fields` on the left of the figure compute derivatives based on the electric field (that depends on time). Their outputs are then used by the `velocities` groups of tasks to compute the velocity field. The `periodic cond.` tasks generate the $f5d'$ field based on $f5d$ with added periodic conditions. All these tasks: `fields`, `velocities` and `periodic cond.` support data parallelism at the granularity of a single point.

Further right in the figure, the `borders` tasks generate some values at the boundaries of the buffers to prepare the computation of the spline coefficients. The `coefs` tasks compute the actual spline coefficients by solving linear problems. The `feet` tasks compute a second order approximation of the coordinate of the origin point by combining the velocity and its spline approximation. Finally, the `interpolate` group of tasks combine these coordinates with the spline representation of the distribution function to evaluate its value at the next time step. All these tasks: `borders`, `coefs`, `feet`, `interpolate` work on a full (r, θ) 2D plane but some of them can be parallelized at a finer grain. The `borders` and `coefs` tasks are very difficult to be efficiently parallelized at a finer grain either due to full plane access such as with linear problem solving or boundary conditions. The `feet`, `interpolate` are challenging to be parallelized at a finer grain either due to complex neighborhood access, boundary conditions or value-driven data indirections on input (r, θ) 2D planes. Both task groups support the write of their output data at the granularity of a single point, but they either require a subpart or a full input (r, θ) 2D plane. Consequently, current task implementations simplify the computation by working at the (r, θ) 2D plane granularity since many tasks require a full input (r, θ) 2D plane. This analysis provides information on the intrinsic properties of the algorithm in terms of task decomposition, data parallelism or dependencies. As we will show in the following section however, some transformations need to be applied to this graph to make it efficient to execute by StarPU.

3.3. Implementation choices and evaluation

In this section, we first present a straightforward StarPU implementation of the previously described spline-based advection DAG. This naive implementation leads to a rather inefficient execution and we therefore propose and implement several incremental optimizations. We evaluate each of these on two datasets and analyze their impact. We also implement and evaluate a task version of the Lagrange-based advection using the same set of optimizations.

Performance evaluations are done on a node of the Poincare cluster (IDRIS, France). Each node contains 2 Intel Xeon E5-2670 (2.60GHz) sockets with 8 cores each. Experiments have been done 20 times and the median is displayed with the minimum and maximum as error bar. The compiler used is Intel ICC 15.0.0 (the latest version available on the machine) with its native OpenMP runtime.

Figures 2 and 3 present the completion time of each version of the spline-based advection on a small ($Nr \times N\theta \times N\varphi \times Nv_{\|} = 64 \times 129 \times 32 \times 16$) and more realistic ($Nr \times N\theta \times N\varphi \times Nv_{\|} = 512 \times 513 \times 32 \times 16$) dataset respectively. The *reference* version has been directly extracted from GYSELA and rewritten as a standalone C++ code. Similar to the original code, the parallelization relies on the OpenMP fork-join paradigm and uses a single *parallel for* to iterate over (r, θ) plans of the buffers using 16 threads. The code is close to a C code and does not use C++ specific features except for arrays that are stored in a dedicated class implementing the parenthesis operator to make multidimensional array access close to the Fortran syntax. The performance of this version is comparable to the original GYSELA Fortran code.

The *intermediate* version uses the same OpenMP parallelization but changes the code structure compared to GYSELA to make introducing tasks easier. The code of this version has been "flattened". *I.e.* the original code had a deep call structure where each subroutine could contain loops, call other subroutines and provide multiple alternative implementations through conditionals. Instead, in this new version, the loops and conditionals have been moved to the outermost subroutine that thus mostly contains a single loop nest. At the heart of each loop, the computation is implemented by a call to another dedicated subroutine that does not make further function call (assuming that compilers are able to inline such functions). Transfer of information through global variables

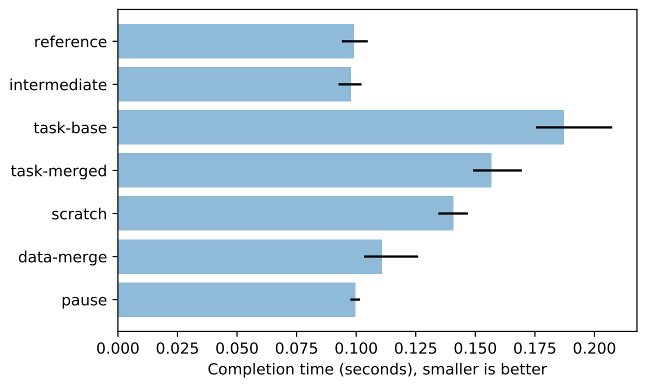

Figure 2. Completion time of all the versions of the spline-based advection on a small dataset ($64 \times 129 \times 32 \times 16$)

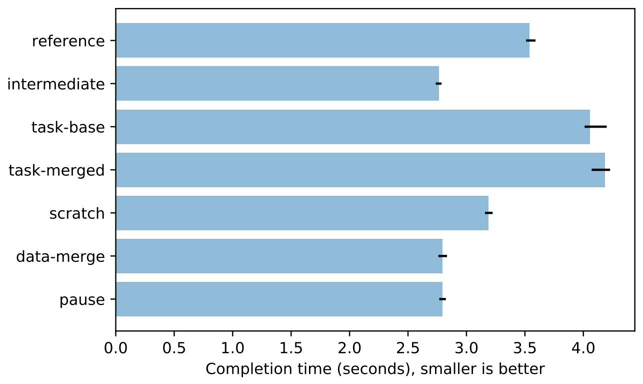

Figure 3. Completion time of all the versions of the spline-based advection on a realistic dataset ($512 \times 513 \times 32 \times 16$)

has also been removed and replaced by function parameters. All in all, these changes make understanding the flow of information easier just by looking at the outermost function. Each function it calls corresponds to one task of Figure 1, the parameters of these functions correspond to the inputs and outputs of the tasks and the loop nest exhibits the available parallelism. The performance of this version is similar to the reference on the small dataset but 22% faster on the realistic one because the simpler code flow is easier to optimize for the compiler.

The *task-base* version builds on the *intermediate* version and replaces the OpenMP parallelization by calls to the StarPU API using 17 threads (1 "main" thread + 16 worker threads). This implementation is 80% slower than the *intermediate* version on the small dataset and 45% on the realistic one. It is also 15% slower than the reference on the realistic dataset even if it includes the optimization contained in the *intermediate* version. This performance degradation is caused by many combined factors that we will handle step by step to understand their impact with each of the following versions.

A first potential source of overhead is the large number of tasks submitted compared to their duration. Indeed each task submission and its scheduling has a cost and if this is not hidden by the task execution cost, the overhead can become important. In addition, having distinct tasks that work on the same data but can be executed on distinct cores can lead to bad cache behavior. The *task-merged* version thus merges all the tasks computing the velocity that read same constant data. This works well on the small dataset where it improves performance by 15% over the *task-base* version but not on the realistic dataset where it is 3% slower. Indeed, for this dataset, the duration of each task is higher and reducing the number of task limits the choice available to the scheduler with only a limited impact on submission overhead.

Another potential source of overhead comes from the management of StarPU "temporary" buffers that are allocated and freed before and after each use. The *scratch* version thus replaces temporary buffers by "scratch" buffers that are allocated only once by thread for the whole duration of the DAG execution when possible (*i.e.* when the buffer is not transmitted between tasks). This significantly improves the performance with a 10% gain on the small dataset and 24% on the realistic one compared to the *task-merged* version. As a result, this version is still 42% slower than the reference on the small dataset, but already 10% faster on the realistic one. The *intermediate* version does however remain faster even on the later case.

The merge of tasks in the *task-merge* optimization makes it possible to go further with the reduction of temporary buffers; not in term of size, but in term on number. Indeed, since each group of four tasks are merged into one task, the four output buffers required by the four tasks can be merged together to a unique output buffer per merged task. The *data-merge* version thus aggregates the four buffers produced by the *fields* tasks into a single one and reduces by four the amount of allocated temporary data. This improves performance by 21% on the small dataset and by 12% on the realistic one over the *scratch* version. On the small dataset, this is still 12% slower than the reference and *intermediate* versions, but on the realistic one, this is comparable to the *intermediate* version, *i.e.* 28% faster than the reference.

A last potential source of overhead we have studied is the contention between application threads and workers for the access to the DAG representation. The *paused* version tries to solve this by stopping the workers during task submission. This optimization works well on the small dataset where many small tasks that imply lots of access to the DAG are submitted. The performance is improved by 10% over the *data-merge* version to reach the same performance as the reference. On the realistic dataset however, it has no noticeable impact.

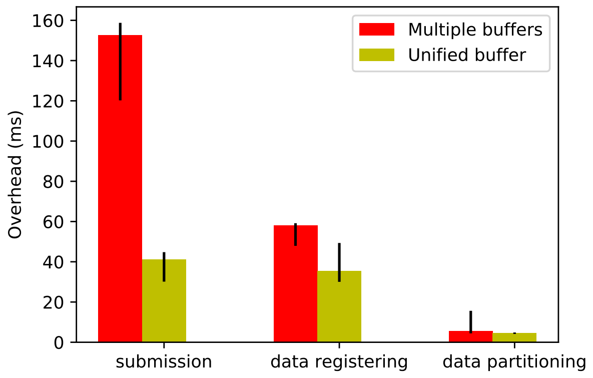

FIGURE 4. Data management overhead analysis for the spline variant on a realistic dataset ($512 \times 513 \times 32 \times 16$)

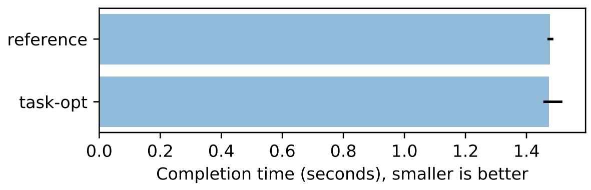

FIGURE 5. Completion time of the reference and task-based Lagrange variant on realistic dataset ($512 \times 513 \times 32 \times 16$)

As one can see on Figures 2 and 3, reducing the amount of StarPU temporary buffers (change from *task-merge* to *data-merge*) has a large impact on performance. On the realistic dataset, this is even the only optimization required to go from the worst case to the best one. In order to better understand this aspect, let us first describe the code-flow associated with temporary buffers in StarPU.

These buffers must first be registered with StarPU to get a handle that can be used as input or output of a task. A single buffer is however typically used in parallel by multiple tasks and one thus typically uses the StarPU partitioning API to get a collection of handles that stand for sub-parts of the buffer (such as a line or a block of a multi-dimensional array). When submitting a task that uses a temporary buffer (or a part of it), StarPU allocates the whole associated buffer. Only when no more task requires the buffer can it be un-registered to let StarPU deallocate the memory.

Figure 4 breaks down the impact of the reduction of temporary data buffers on the overhead introduced by the task submission thread. It respectively shows the cost of task submission, data registering (which excludes unregistering) and data partitioning. In the *scratch* version (which uses multiple temporary data buffers), the major overhead source comes from submission of tasks. Indeed, task submission causes the immediate costly allocation of temporary data needed to compute the task. The *data-merge* version strongly reduces the overhead by reducing the number of allocated buffers. This version also reduces the cost of data registering since less temporary buffers are created.

The use of a submission thread besides the 16 worker threads on a 16 cores node can result in bad performance. The impact strongly depends on the amount of work done by the submission thread (*i.e.* the number of registered and partitioned data as well as the number of submitted tasks). When this thread is overloaded, it becomes appropriate to use 15 workers threads rather than 16. However, the impact of the submission thread is negligible in this kernel.

Finally, in order to validate this optimization approach, we have applied the exact same process for the Lagrange variant of the 2D advection: tasks and temporary buffers have been merged, scratch data are used and the workers are stopped during task submission. The granularity used on this variant is still (r, θ) 2D planes. Even though the submitted task graph differs (only the left part remains: the `fields` tasks), it is still compliant with Algorithm 1. Figure 5 presents the execution time for the OpenMP reference and the fully optimized task version on a realistic dataset ($Nr \times N\theta \times N\varphi \times Nv_{\|} = 512 \times 513 \times 32 \times 16$) using 16+1 threads. On this variant of the code with no dedicated study, the performance of the task implementation is comparable to the reference.

3.4. **Discussion**

The introduction of tasks was performed in three steps. First, the reference code has been adapted to focus on the flow on data in order to fit well with task-based approaches, leading to the intermediate version. This means that the code should be most of the time redesigned according to the tasking model. Then tasks have been added in the code according to the heavy-grained parallelization performed by the current 2D advection code. Finally, multiple optimizations have been successively devised to ultimately provide high-performance regarding the efficiency of intermediate version.

Experimental results show that task implementation is competitive with the reference implementation, comparable on the Lagrange variant and up to 27% faster on the spline variant. However, special attention should be paid on data management in order to limit the overhead introduced by data allocation and data registering to the runtime, both in term of performance and memory overhead [15]. This has been done using scratch data and by adapting the temporary data granularity. Moreover, task granularity should also be carefully chosen since it is closely related with the number of temporary data. Indeed, splitting a task in two or more involves the use of temporary data to assure information sharing between the tasks. Finally, it can sometimes be relevant to stop the workers during the task submission.

One must however notice that the best optimizations strongly depend on the dataset size. This is because tasks operate on a too small granularity for small datasets, which make some overheads noticeable that would not be apparent otherwise. It is really important to adapt the optimizations process and task granularity to realistic data sizes to reach good performance in production.

Both the original GYSELA code and the scheduling of the task version presented here rely on a coarse-grain parallelism handling a full (r, θ) 2D plane by thread. To leverage finer-grain parallelism, the algorithm would enable processing one or two planes at a time which typically fits in cache, instead of one by thread that typically does not. A task based approach is needed to leverage this fine grain parallelism since it requires having one thread working on the rightmost part of the DAG of Figure 1 while the others work on the left part; *i.e.* the threads do not execute the same code anymore. The task version implemented here is therefore a promising first step towards a version of the code with better cache behavior that will be the focus of some future work.

StarPU codes described and evaluated in this section leverage intra-node computing power. StarPU also enables to schedule tasks graphs on multiples nodes and bring a way to overlap computations and communications. This can be done by replicating the same task graph on multiple nodes and by interleaving computation task scheduling with asynchronous MPI send/recv communication task scheduling. However, the evaluated kernel is mainly compute-bound with a uniform workload between nodes. Thus, we expect performance gains to be the same on distributed versions of the StarPU codes. Consequently, this paper only focuses on the intra-node evaluation since there is a room for improvement in intra-node versions.

The current StarPU implementation enables the adaptation of the task graph scheduling according to the targeted hardware. Thus, it decouples the problem of writing the kernel algorithm from finding an efficient execution order on the underlying platform. As a result, it becomes easy to switch between a depth-first execution and a breadth-first one, or even a mixed approach in order to improve cache reuse. This might prove especially useful for architectures with deep and complex cache hierarchies such as on the Intel Xeon Phi. This is however not the focus of this work and thus left as future work.

While the dynamic scheduler provided by StarPU can make automatic optimization choices when it comes to task-grain parallelism, this section has demonstrated that the task should not be too fine grained for its overhead to remain reasonable. This means that static finer-grain optimizations such as vectorization, blocking, etc. are required inside the implementation of tasks. The following section focuses on this aspect for the `interpolate` task, the rightmost one in the DAG of Figure 1.

4. Kernel-level auto-tuning

Automatic performance tuning is an empirical feedback-driven performance optimization technique that aims at shortening execution time across a wide set of systems and architectures together with good software portability. Auto-tuning is based on three pillars: designing a meta-program that includes architecture-relevant optimizations and their associated parameter space, automating the generation of effective code for this optimization space, efficiently searching within this optimization space through compiling/executing pieces of the generated code.

Structuring application code around small computation kernels is becoming quite common nowadays. Several libraries and framework are promoting such approach for software engineering within HPC applications. Typically, a kernel is one or several functions for which inputs and outputs are well identified, for example in order to transmit data by specific mean or efficient mechanisms. Also, the kernel can possibly be written in a specific language, as, for example in: CUDA, OpenCL, BOAST. It permits to exhibit clearly what is the portion of the code that is compute or memory bound and subject to parallelization and optimization. The rest of the application code, apart from the kernel, is considered as less greedy in terms of execution time.

4.1. **Auto-tuning using BOAST**

One of the main goals of the BOAST (Bringing Optimization Through Automatic Source-to-Source Transformations[2]) framework is to design computing kernels that present good performances on diverse architectures used by the application for production, while still being portable after the optimization process took place. Indeed, architectures are evolving and changing quite fast in HPC field compared to the lifetime of an application. Investing manpower to optimize the application for a new architecture is quite reasonable, suffering hindrance from previous optimization work is not. Thus optimizations have to be as orthogonal as possible from one another so as to be easily revised to fit a new architecture.

BOAST also allows to combine small computing kernels into bigger ones either by calling them or inlining them. Thus, a bottom up approach can limit the size of individuals kernels and allow independent optimizations of small kernels. This is a way to reduce the search space used for the auto-tuning process as well as modularizing the implementation. Optimizing individual loop nest can also be a viable strategy. Another point that BOAST handles is helping the programmer to deal with huge optimization space to search for auto-tuning.

In the remaining of the section we will briefly present BOAST's syntax. BOAST is an Embedded Domain Specific Language (EDSL) written in Ruby. A more in-depth tutorial is available in Annex A, or in the BOAST wiki[3]. The BOAST documentation[4] gives more details on its usage.

```ruby
decl i = Int("i")
decl f = Real("f", :dim=>[Dim(4), Dim(4)], :local=>true)
```

```c
int32_t i;
double f[4*4];
```

```fortran
integer(kind=4) :: i
real(kind=8)    :: f(4, 4)
```

Listing 1. BOAST, C and Fortran code for declaring variables

An example of implementing a kernel in BOAST is shown in Listing 3. Variables are declared with the `decl` keyword. Listing 1 shows equivalent declarations for a scalar and vector variable in BOAST, C and Fortran. The `pr` method writes the code passed to it to the current kernel. For instance, `pr c === a + b` will generate `c = a + b` when generating Fortran code. Notice in this case the assignment must use three = characters in order to differentiate between Ruby and BOAST assignments. The other classical algebraic and logic operations are also available with the same syntax as the addition.

[2]https://github.com/Nanosim-LIG/boast

[3]https://github.com/Nanosim-LIG/boast/wiki/Lagrange1d

[4]http://www.rubydoc.info/github/Nanosim-LIG/boast

The use of `Case`, `If` and `While` in BOAST is equivalent to their use in both C and Fortran. `For` however, behaves closely to Fortran: it requires the variable to be iterated over, an initial value, the final value (inclusive) and an optional step. For example, `For(i, 0, 10)` is equivalent to `for(i = 0; i <= 10; ++i)` in C, or `do i = 0, 10, 1` in Fortran.

```
n    = Int( "n",    :dir=>:in)
inpt = Real("inpt", :dir=>:in,
            :dim=>Dim(0, n-1))
rslt = Real("rlst", :dir=>:out,
            :dim=>Dim(0, n-1))
pr p = Procedure("foo", [n,inpt,rslt]) {}
```

```
void foo(
  const int32_t n,
  const double * inpt,
  double * rlst) {}
```

```
SUBROUTINE foo(n, inpt, rlst)
  integer(kind=4), intent(in) :: n
  real(kind=8),    intent(in) :: inpt(0:n-1)
  real(kind=8),    intent(out):: rlst(0:n-1)
END SUBROUTINE foo
```

Listing 2. BOAST, C and Fortran code for creating a subroutine.

Functions and procedures are created with the `Procedure` control structure. A `Procedure` needs two arguments: the name of the procedure, and a list of parameters. An example is shown in Listing 2, along with the resulting C and Fortran code.

The `build` method of the kernel is used to generate and compile code in the specified language and with the given `CFLAGS`, `FCFLAGS` or other configurations. The `run` method of the kernel will build it if necessary, execute the kernel with the given arguments, and return a dictionary with the return value if the kernel is a function, scalars that have been passed by reference and modified, execution time and, if available, energy consumed.

4.2. Optimizing GYSELA with BOAST

4.2.1. *Integration process of the 2D advection kernel into BOAST*

We have first extracted four subroutines from GYSELA that implement substeps 3 and 4 of Algorithm 1 described in Section 2.2. This constitutes the kernel that we intend to improve with BOAST. The API of these subroutines has been changed a bit in order to simplify the BOAST implementation. Indeed, transmitting complex data structures is possible but quite wordy for BOAST. Then, as it was possible to do so, we restricted all parameters of our subroutines to be scalars or multi-dimensional arrays with types of integer or float (64-bit).

In order to integrate these four subroutines into the BOAST framework, we began with the most internal one in order to have no dependency towards undefined subroutines. Then, we added incrementally the three other subroutines into BOAST framework. Each time a subroutine was added, we provided a corresponding robust regression/unit test. For this issue, it was required to establish valid fake inputs for each subroutine, and also to give methods that can check the outputs of the subroutine for these inputs. These strict checks are required in order to get a valid kernel at the end of the auto-tuning process.

Once we have finished defining the Ruby description of the subroutines and of the unit tests, BOAST can be used to produce such outputs: a Fortran file containing four subroutines or a C code containing four functions or two object files corresponding to compiled source files.

In term of working hours, integration of this kernel into the BOAST framework has required to mainly invest time in: learning a bit of Ruby and BOAST syntax, but also in redesigning the routines to incorporate the parameters subject to the auto-tuning process. One of the trickiest parts for us was the quest for meta-program parameters that necessitates different kinds of skills: a good understanding of the considered kernel, compiler optimization knowledge, BOAST syntax ability.

4.2.2. *Sample extract of BOAST code*

Listing 3 shows part of the BOAST code for the 2D advection kernel. Within this extract of the kernel, a loop tiling is providing vectorization opportunities to the compiler. At the beginning, some statements declare small arrays of size `vector_size`. Then, the first loop of Part 1 iterates on a set of these small arrays. In the loop nest, the feet of characteristics are computed for sets of `vector_size` points (first function call), and the interpolation at these feet is derived shortly after (second function call). Further, the Part 2 proceeds

```
# Declarative part
decl vrfeet      = Real("vrfeet",  :dim => Dim(0,vector_size-1), :align => 64)
decl vthfeet     = Real("vthfeet", :dim => Dim(0,vector_size-1), :align => 64)
decl fres        = Real("fres",    :dim => Dim(0,vector_size-1), :align => 64)
decl irend_vecto = Int("irend_vecto")
decl mod_vecto   = Int("mod_vecto")
decl ir          = Int("ir")
decl cc          = Int("cc")

pr mod_vecto === mod(ir_end-ir_start+1,vector_size)
pr irend_vecto === ir_end - mod_vecto

# Part 1: Loop blocking with parameter vector_size
pr For(ir, ir_start, irend_vecto, :step => vector_size) {
  pr p_bsl_2d_lag_taylor_onefoot_vect.call(...,vector_size,vrfeet,vthfeet)
  pr p_lag_interpolation_rtheta_vect.call(...,vector_size, vrfeet, vthfeet, fres)
  pr For(cc, 0, vector_size-1) {
    pr fout[ir+cc,itheta,iphi,ivpar] === fres[cc]
  }
}

# Part 2: Remaining portion of the loop
pr For(ir, irend_vecto+1, ir_end, :step => 1) {
  pr p_bsl_2d_lag_taylor_onefoot.call(..., vrfeet[0],vthfeet[0])
  pr p_lag_interpolation_rtheta.call(...,vrfeet[0],vthfeet[0], fres[0])
  pr fout[ir,itheta,iphi,ivpar] === fres[0]
}
```

Listing 3. Part of the BOAST code for the 2D advection - loop tiling

with the rest of the loop indexes in order to reach the final index `ir_end` (called remainder loop). This last piece of code is executed only in the case the total number of indexes is not a multiple of `vector_size`. Typically, subroutine calls inside Part 1 will be possibly vectorized by the compiler. The integer `vector_size` will play a central role here because this length may authorize good utilization of vector registers. It will be the subject of the BOAST auto-tuning procedure in order to determine the best value of this parameter for a given compiler/architecture.

4.2.3. *Auto-tuning parameters*

The set of tunable parameters in the BOAST meta-program are of utmost importance in order to reach better performance out of the auto-tuning process. Hereafter, we will go through the main tunable parameters that we have defined in the advection kernel. This list can be extended if we find other opportunities that accelerate computations.

- **Targeted language.** Within GYSELA, Fortran 90 is the most used language, but the C language is also employed. Therefore, it is worthwhile to evaluate the opportunity to generate and compile C or Fortran and link the resulting object file to the GYSELA executable.
- **Loop unroll.** BOAST provides a way to parameterize the loop unrolling. Specific locations where we found that unrolling impacted significantly the execution time have been equipped with BOAST unrolling.
- **Inlining.** For the four subroutines that have been written into the meta-program, two alternatives have been used to inline inner-most subroutine calls. The first solution is to tell to the compiler through a directive to inline a given call (*e.g.* `!DIR$ FORCEINLINE` with INTEL Fortran). The second solution is to make use of the builtin feature of BOAST to produce this inlining.
- **Intrinsics.** BOAST can produce INTEL intrinsics (see paragraph 4.4.1 that follows). We have written a version of the meta-program that generates directly a code close to an assembly code. We can use or this kind of explicit vectorization, or, as a backup solution, a higher level of expression.
- **Block size.** As mentioned in Section 4.2.2 and Listing 3, loop tiling is hardwired in the meta-program. Auto-tuning process will go through the different values of the blocking parameter of the loop (that

interacts with vectorization done by the compiler). Compiler will optimize differently depending on block size value.

- **Compiler version.** On modern supercomputers, a set of compilers is often available. A scan over the versions of `GNU Fortran`, `INTEL Fortran`, `GNU gcc`, `INTEL icc` is useful in order to identify which compiler achieves the lowest execution time for a given kernel.

The final BOAST meta-program is composed of 200 lines that correspond to the substeps 3 and 4 of Algorithm 1. It should be compared to the original Fortran code of 300 lines. Thus, the meta-program that includes much more information than previous code is also significantly shorter.

4.3. Getting the optimal kernel

4.3.1. *Benchmark*

We have applied the BOAST auto-tuning process on four different architectures in order to examine what are the gains brought by selecting the best generated kernel. Rheticus machine (Figure 6, Intel X5675, 3.07GHz, 12-cores node) is hosted at University of Aix/Marseille-France. Poincare machine (Figure 7, Intel E5-2670 v1, 2.60GHz, 16-cores node) is located at IDRIS, Orsay-France. The third machine, Occigen is at CINES, Montpellier-France (Figure 8, Intel E5-2690 v3, 2.60GHz, 24-cores node). The fourth machine, an INTEL KNL node, is also at CINES (Figure 9, Intel 7210 1.30GHz, 64-cores node) and its MCDRAM memory is configured as quadrant-flat.

The size of the computational domain is relatively small: $Nr \times N\theta \times N\varphi \times Nv_{\|} = 256 \times 1 \times 1 \times 1$. This setting is not representative of a realistic GYSELA run. However, we intend to use the BOAST auto-tuning approach to optimize a single computation intensive kernel. Then, the test case has been chosen so that the set of data manipulated fit into the L2 cache of each of our target machines in order to focus on optimization of computations. Only the size of the dimension along r is representative of standard runs. This hot-cache setting avoids any trouble associated to memory transfers and cache effects and may reduce standard deviation in the time measurements. The MCDRAM mode of the KNL machine has therefore no impact on the benchmark measurements.

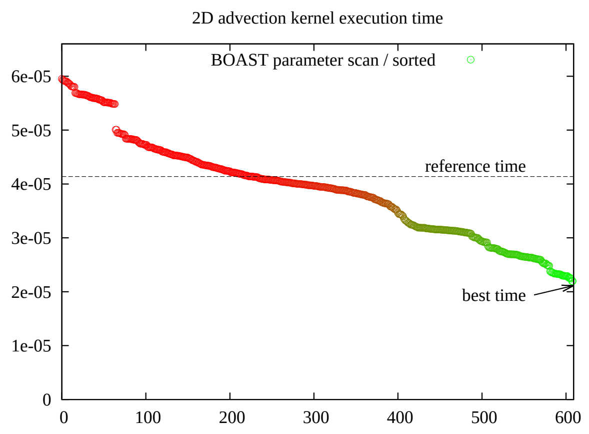

FIGURE 6. Rheticus machine - Westmere - Exec. time in s. (ordinate) of the advection kernel, abscissa is a sorted index - each index represents a set of BOAST parameters

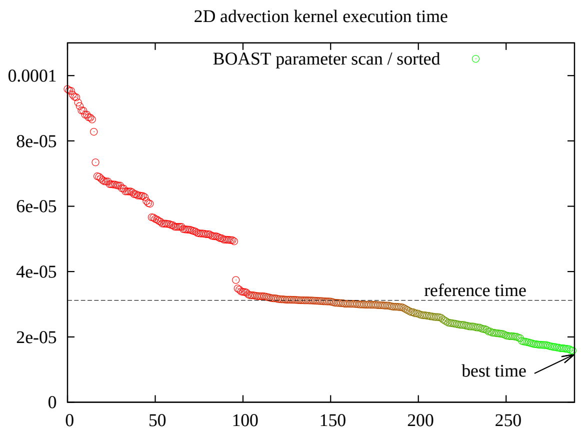

FIGURE 7. Poincare machine - SandyBridge - Exec. time in s. (ordinate) of the advection kernel, abscissa is a sorted index - each index represents a set of BOAST parameters

Figures 6, 7, 8 and 9 show execution times over the whole parameter set scanned by BOAST. The printed execution time is an average over thousands of kernel executions. The parameter space is represented in

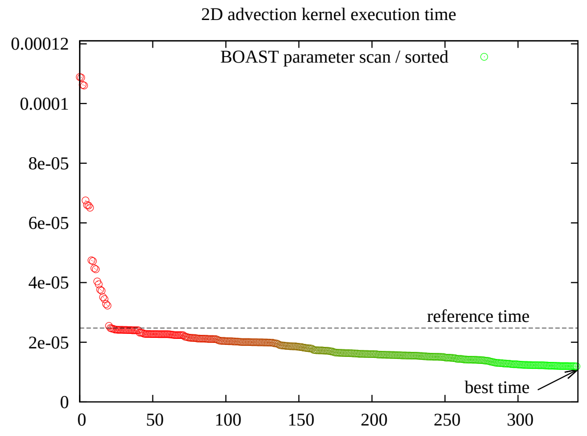

Figure 8. Occigen machine - Haswell - Exec. time in s. (ordinate) of the advection kernel, abscissa is a sorted index - each index represents a set of BOAST parameters

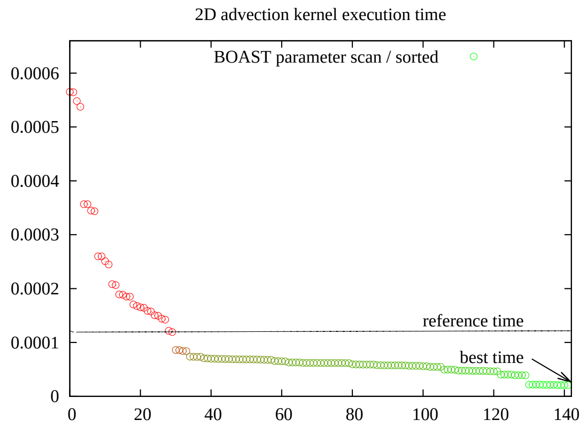

Figure 9. INTEL KNL - Knights Landing - Exec. time in s. (ordinate) of the advection kernel, abscissa is a sorted index - each index represents a set of BOAST parameters

abscissa, each index corresponds to one experiment characterized by a set of parameters. One point for example corresponds to the parameters {`lang=FORTRAN, unroll=false, inlining=Compiler, intrinsics=false, block.size=1, compiler=gfortran4.9`}. In order to improve readability, the points have been sorted in descending order of execution times. Furthermore, to stress the fact that the slower timings are better than the higher timings, red color circles have been used for large timings, whereas green color circles have been used to plot small timings. The best generated kernel therefore corresponds to the rightmost point (lower is better for execution time). A reference initial execution time was obtained with the original Fortran code, it is represented by an horizontal dashed line. The speedup brought by auto-tuning can be directly obtained by taking the horizontal dashed line (reference execution time) divided by the ordinate of rightmost position point (best configuration).

	Machine			
	Rheticus	Poincare	Occigen	KNL node
lang	FORTRAN	FORTRAN	FORTRAN	FORTRAN
unroll	true	false	false	false
inlining	BOAST	Compiler	Compiler	BOAST
intrinsics	false	false	false	false
block.size	**4**	**32**	**4**	**32**
compiler	**intel/16.0.2**	**intel/13.0.0**	**intel/14.0.4.211**	**intel/17.0**
Execution time	22.0 μs	15.8 μs	11.9 μs	20.8 μs
Speedup	**1.9**	**2.0**	**2.1**	**5.7**

Table 1. Best parameter set on the testbed, and speedup of execution time over the reference time obtained with the default INTEL compiler and the original Fortran code.

On the set of four machines, the auto-tuning process gives the best timing for the set of parameters given in Table 1. On each tested machine, the *Fortran* language wins over *C* language and the INTEL compiler gives shorter execution time than the GNU compiler (especially for Fortran Language). Also, the option that produces *intrinsics* through BOAST code generation is not efficient enough to beat the standard approach with vectorization handled by the compiler, even if we tried hard to find several combinations to use vector registers the best way we can. It is quite disappointing for our intrinsics version but it means that vectorization made by the compiler on the non-intrinsics version is quite competitive. The two parameters *inlining* and *unroll* have not a single value through all the machines, the auto-tuning process is helpful to extract the best choice.

The *compiler version* has a major impact on performance and surprisingly depends also on the considered machine. The *block size*, which is the loop tiling parameter, is a complex parameter as its impact on the performances depends highly on the compiler used. Figure 10 shows the impact of the *block size* on the different platforms using the different *compilers*. First, `GFortran` does not optimize the code well and it is mandatory to help the compiler with hand-tailor loop tiling. But still `GFortran` is not as efficient as the other compilers to optimize the code and for which the loop tiling is far from critical. Second, there are some correlation between the *block size* and the *compiler*, because the behavior of the *block size* is similar between platforms and for a given compiler. For `GFortran` the impact of the *block size* across the platforms can be modeled with $(blocksize + \frac{1}{blocksize})$ as shown in Figure 11. However for the other compilers, even if we could identify a general tendency, the previous model does not fit correctly, because the measurements are much more noisy (in the sense that the parameters which are not accounted for in the model have a non negligible impact). This noise could be explained by the fact that `ICC`, `GCC` and `IFortran` are able to perform more complex optimization and thus are less predictable. In this kind of situation, auto-tuning really makes sense because it is not possible to guess correctly which optimization to put into place in the code and how to tune it correctly to handle all architectures and compilers.

Comparing Figure 6 to 9, it is quite remarkable that the ratio of execution time of the slowest over the fastest execution times are in-between 3 and 27. In addition, the speedup of fastest execution time over reference execution time is between 1.9 and 5.7. These facts mean that auto-tuning really leads to a large benefit and helps producing an optimized version of the original kernel.

It is also quite interesting that on Figure 8 and 9, the curve exhibits a quite sharp gradient at leftmost position. It means that several configurations lead to serious trouble in terms of performance. Then, auto-tuning gives us an adapted procedure to avoid such kind of bad settings.

Table 2 gives some useful insight on the relative significance of each parameter of the meta-program. Considering each machine of the testbed, we fix one single parameter (*e.g.* `lang` in the first row with the setting `F90`), and we explore all possible values of all other parameters except `lang`. We then sort all obtained execution times, and we retain the best one. Then, we fix another value of this single parameter (*e.g.* `value2=C`), and find also the best execution time. Finally, we print out the increase in term of execution time between `value1` and `value2`. We have chosen `value1` that matches the best configuration of Table 1. This rationale explains why we always have an increase of execution time whenever considering `value2` different to `value1`'s reference. The largest the increase, the more significant the parameter for the considered machine is. Practically, we have provided the `value2` in the Table that provides the largest increase, indeed `compiler` and `block.size` has more than two possible values.

Interpreting this Table, we learn that the less important parameters are `unroll` and `inlining`. We note the Fortran language brings a substantial benefit over C on all machines, and also the version with no-intrinsics wins the one with intrinsics. The two main parameters that we have to focus on are the compiler version and the block size for the vectorization, they both have a big impact on all considered machines. BOAST auto-tuning mechanism permits to select the best choice.

4.3.2. *Integration in the full* GYSELA *code*

The BOAST version of the kernel offers great performance and performance portability. This improvement would although be of little interest if not used in the production runs of the GYSELA code. Integration in the full code does, however, have to take into account not only performance but other aspects also such as readability and usability. GYSELA is indeed the result of a collaboration between people coming from different scientific fields –mathematics, physics, computer science– and with various expertise levels concerning software development. The code can only keep evolving if it remains possible for all these people to use it and understand the semantic it implements.

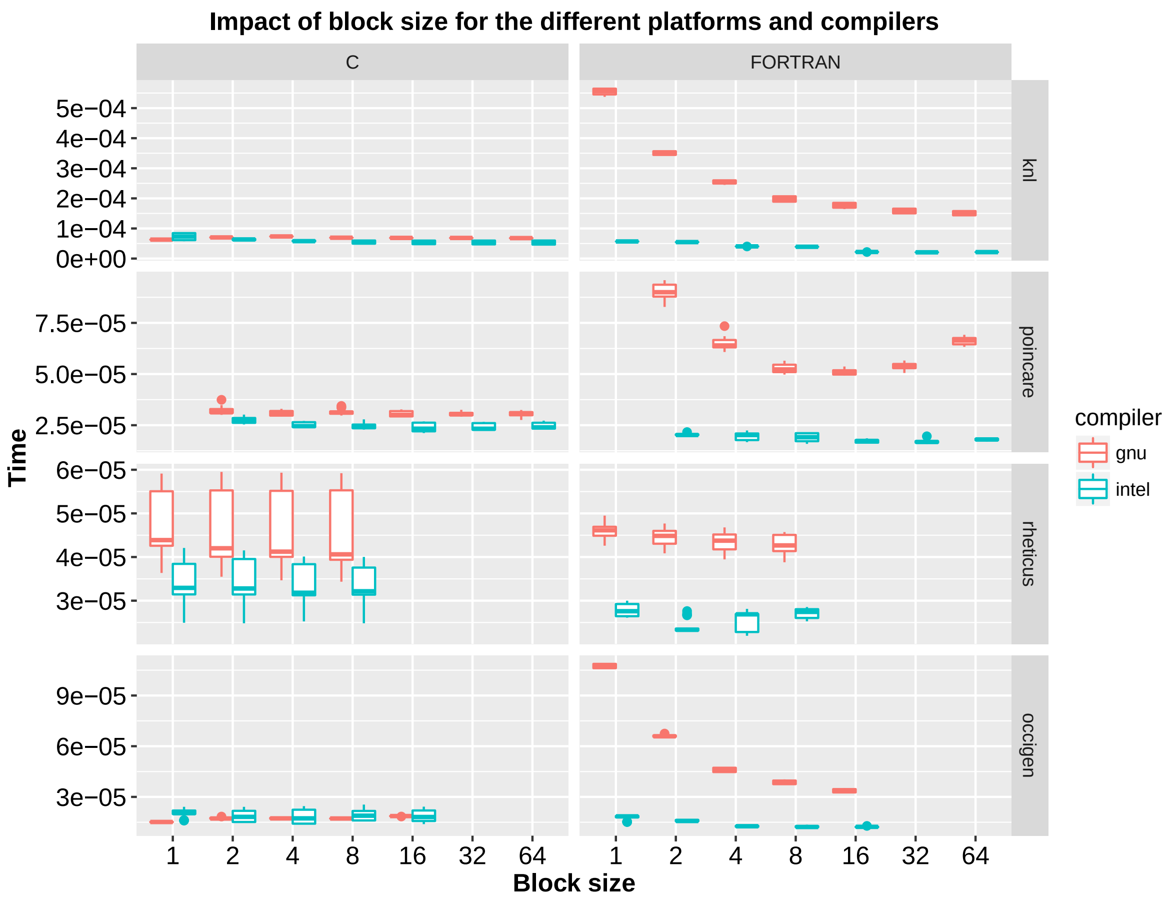

FIGURE 10. The impact of block size differs from a compiler to another but for a given compiler, the model for block size, is similar across the different platforms except for the Rheticus platform. Which means there are some correlation between the block size and the compiler / language.

On the bright side, one should recall that the number of lines of the BOAST kernel is less than the original Fortran version (200 lines vs. 300). This does however not mean that it is simpler to understand but merely reflects the relative compactness of the Ruby-based BOAST DSL in comparison to the verbosity of Fortran. The BOAST version of the kernel does indeed add a layer of logic where potential optimizations are explicitly specified on top of the original kernel and thus buries the semantic deeper. In addition, while language such as Fortran, C, Python or Matlab are rather widespread in the scientific simulation community, knowledge about Ruby and furthermore the BOAST DSL would have to be acquired by most with the single purpose of understanding these kernels. The solution we have adopted is to keep a simple Fortran version of the kernel in the code. It is not intended to be used in production runs but eases understanding the code semantic. We have also developed a script that extracts this simple version of the kernel from the code thanks to annotations in the source. We use it as reference for regression testing in the BOAST auto-tuning process. This ensures that the simple Fortran and BOAST versions of the kernel remain synchronized in terms of produced results. We have finally adapted the build-system so that switching from one version of the kernel to the other can be specified with a simple option. This enables people with no knowledge of BOAST to still study and modify the parts of the code that have been re-written in this framework.

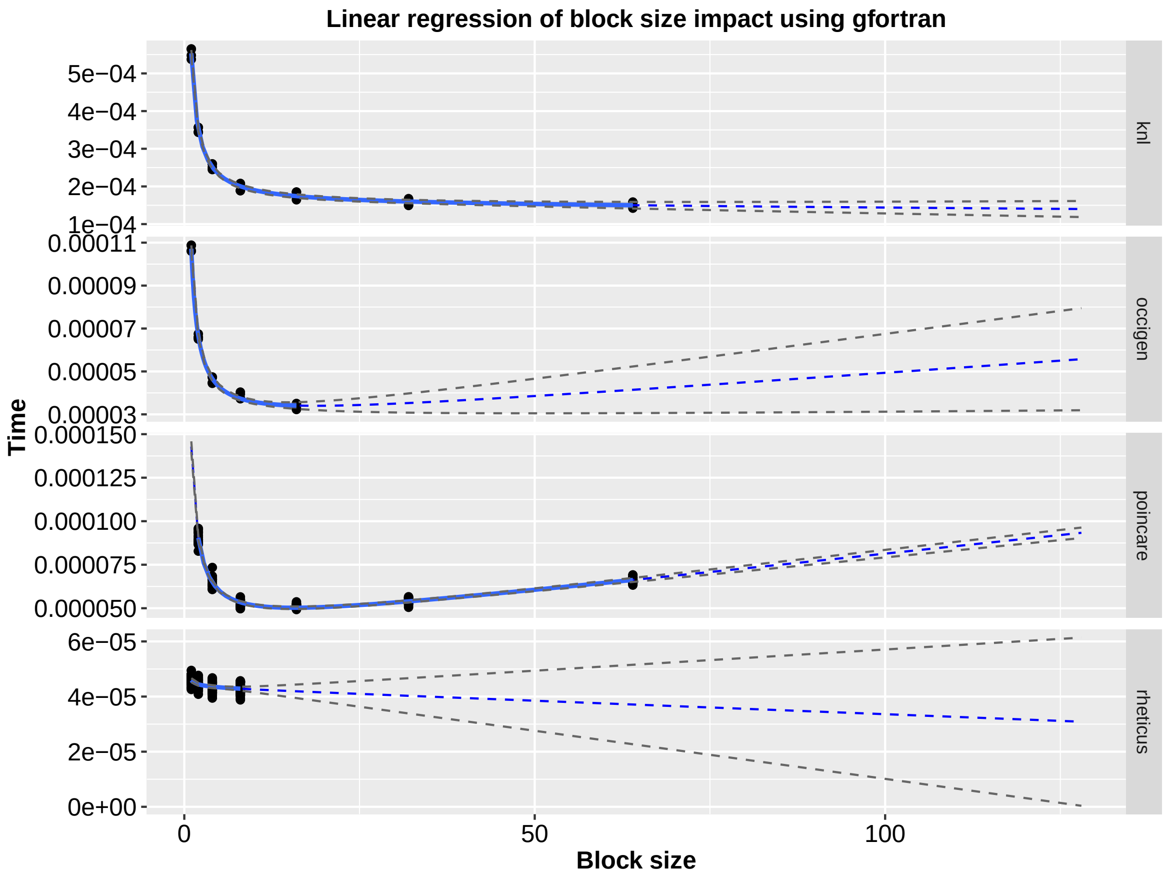

FIGURE 11. The blue line represents the regression line, the dashed blue line represents the estimated values predicted by the model, and the dashed grey lines represent the confidence intervals. With the compiler `gfortran`, the model $(blocksize + \frac{1}{blocksize})$ fits correctly on almost every architecture except on Rheticus where there is lot of noise. We can see that for the platform Poincare the best value for block size was found, but for the other, other bigger values for block size could have been tested. Our linear model could help us to predict the impact block size. However this model is incomplete because it needs additional point to predict the values that have not been tested yet.

This modification of the GYSELA build-system does however lead to a question regarding the generation of the BOAST kernel. Selecting the best kernel is a lengthy process and making it a step of the everyday code compilation would severely impact the duration of compilation. This is not acceptable for a code like GYSELA that is distributed in source form and where each user frequently recompiles its own executable. The good news however is that the code is only run in production on a limited number of machines. Typically the number of distinct machines used is between five and ten with a turnover of one or two every year. We have therefore wrapped the BOAST tool inside a script that splits its use in two steps. The first step determines the parameter values that generate the most efficient kernel and dump these values to a JSON[5] file. The second step compiles the code based on the choices stored in this file. We have chosen to cache the JSON file for each machine in the code repository so as to spare users from having to execute the optimization process at every compilation.

[5]JSON (JavaScript Object Notation) is a lightweight data-interchange format: `www.json.org`

Parameter	best value (value1)	value2 impact on best exec. time	
lang	F90	C	+13%
unroll	True	False	+4%
inlining	BOAST	Compiler	0%
intrinsics	False	True	+13%
compiler	intel-16.0	gnu-4.8	+56%
block.size	4	1	+14%

a) Rheticus

Parameter	best value (value1)	value2 impact on best exec. time	
lang	F90	C	+34%
unroll	False	True	+1%
inlining	Compiler	BOAST	+1%
intrinsics	-	-	-
compiler	intel-13.0	gnu-5.4	+74%
block.size	32	2	+25%

b) Poincare

Parameter	best value (value1)	value2 impact on best exec. time	
lang	F90	C	+15%
unroll	False	True	+1%
inlining	Compiler	BOAST	0%
intrinsics	False	True	+67%
compiler	intel-14.0	gnu-4.9	+25%
block.size	4	1	+25%

c) Occigen

Parameter	best value (value1)	value2 impact on best exec. time	
lang	F90	C	+124%
unroll	False	True	0%
inlining	BOAST	Compiler	0%
intrinsics	False	True	+177%
compiler	intel-17.0	gnu-4.9	+203%
block.size	32	1	+174%

d) KNL

Table 2. Dependency of kernel execution time to a single parameter change. The value of a given parameter is fixed (value1 or value2). The whole domain is still explored for all other parameters and the best time is retained. We evaluate the execution time increase (percentage indicated in color) taking the difference of the best execution time with value2 against the best execution time with value1. Value1 corresponds to the overall best as identified in Table 1.

A limitation of this approach however is that each user is required to install BOAST even if only to execute the second step. We therefore cache the generated source code alongside the parameters so as to enable the compilation of the code without BOAST. The optimization process can thus be executed independently of the compilation, at a much lower frequency, typically only when the kernel has been modified or the machine hardware and software stack has changed. An interesting improvement of this process would be to automate it by using the already existing GYSELA continuous integration platform [6].

One last aspect that we just started looking into is the combination of code compiled using different compilers in a single application. The compiler is one of the parameters identified in the optimization space of each kernel (and an important one as can be seen in Table2). It is therefore possible for a distinct compiler to be selected for distinct kernels written using the BOAST framework. In order to build a single application based on the combination of these kernels with a distinct compiler each, one must look into this issue. As of now, a single BOAST kernel is used in GYSELA and the compiler selected for this one can be used to compile the whole application. In the perspective of using BOAST for a larger part of the code, we have however conducted some preliminary experiments. The Fortran application binary interface (ABI) of the GNU and Intel compilers is luckily compatible as long as one does not use Fortran `modules` which makes mixing code theoretically possible. An even more portable way would be to rely on the Fortran `ISO_C_BINDING` to specify kernel interfaces with an ABI defined by standards. The remaining incompatibilities we have encountered are related to clashes of libraries implicitly included by the compiler. As long as one only relies on the standard C and Fortran libraries embedding them statically in the kernels has solved this for us. This study does however remain very preliminary and we expect more compatibility issues to arise if more features become used in the BOAST kernels, such as OpenMP parallelism for example.

```
require 'BOAST'
include BOAST
set_model(:knl)

a = Real :a, :vector_length => 8
b = Real :b, :vector_length => 8
c = Real :c, :vector_length => 8

decl a, b, c

pr a === b*c + 2.0
pr a === b*c + a
pr a === FMA(b, c, a)
```

```
__m512d a;
__m512d b;
__m512d c;

a = _mm512_add_pd(
        _mm512_mul_pd(b, c),
        _mm512_set1_pd(2.0));
a = _mm512_add_pd(
        _mm512_mul_pd(b, c),
        a);
a = _mm512_fmadd_pd(b, c, a);
```

```
real(kind=8) :: a(8)
!DIR$ ATTRIBUTES ALIGN:64 :: a

real(kind=8) :: b(8)
!DIR$ ATTRIBUTES ALIGN:64 :: b

real(kind=8) :: c(8)
!DIR$ ATTRIBUTES ALIGN:64 :: c

a = b*c + 2.0_8
a = b*c + a
a = b*c + a
```

Listing 4. Vector code in BOAST, C and Fortran

4.4. BOAST improvements

The work on the GYSELA kernel incurred new developments in BOAST in order to support all the target architectures. Evaluating the energy efficiency of the kernels was also of interest.

4.4.1. *Knight Landing Support*

One way to increase the parallelism without increasing the number of cores and the number of execution threads is to increase the parallelism inside the floating point unit. The most common way to achieve this is by computing several elements at the same time. This scheme is called Single Instruction Multiple Data (SIMD). With this approach, the processor manipulates vectors of data rather than scalar values.

Intel processors have seen the number of elements they can process increase during the past 15 years. First, SSE in 1999 allowed to process 2 double precision numbers simultaneously and 4 single precision numbers. Second, AVX in 2011 doubled the vector length. Last, AVX-512 (released in 2016) doubled yet again the vector length. So, using AVX-512 the processor can execute operations on 8 double precision floating point values (and 16 single precision floating point values) simultaneously.

The drawback is that, if neither the programmer nor the compiler manages to use the vector units, you can only achieve a fraction of the peak performance. For instance, on the Knight Landing's architecture (called also KNL) that uses AVX-512, and possesses a peak theoretical performance of 3 TFlops/s (10^{12} double precision floating point operation per second). If one is restricted to scalar computations the peak performance that can be achieved is less than 400 GFlops/s.

As the Knight Landing architecture was one target of the project, one of the necessary improvements to BOAST was thus to add support for the AVX-512 instruction set. We extended the SSE and AVX/AVX2 support to also support AVX-512. BOAST proposes to generate intrinsic C instructions from BOAST vector code. The Ruby vector code can also be used for FORTRAN generation.

Listing 4 shows an example of a BOAST code written using vectors of 8 double precision numbers. The code declares 3 vector variables and then does several small example computations. The last example is a fused multiply add. Listing 4 also shows the generated C and Fortran code.

4.4.2. *Measuring Energy Consumption*

BOAST gives the energy consumed by a kernel execution, if that information is available in the platform. Ultimately the energy data is read from the Intel RAPL energy counters available in several Intel CPUs. These counters give the energy consumed by the whole CPU, core and either memory or integrated graphics card, depending on the CPU model. Since Linux 3.13 it is possible to read these counters from the powercap interface. When possible, BOAST uses powercap. With older Linux versions this interface is not available. In these cases, BOAST reads the values from the model-specific registers (MSR), which requires root permissions. Without these permissions it attempts to read the counters through the likwid daemon, which in turn needs root access but is often granted by the system administrator. The energy data is returned in a dictionary along with the execution time. The energy consumed by the package, core, uncore and memory is given for each socket. This

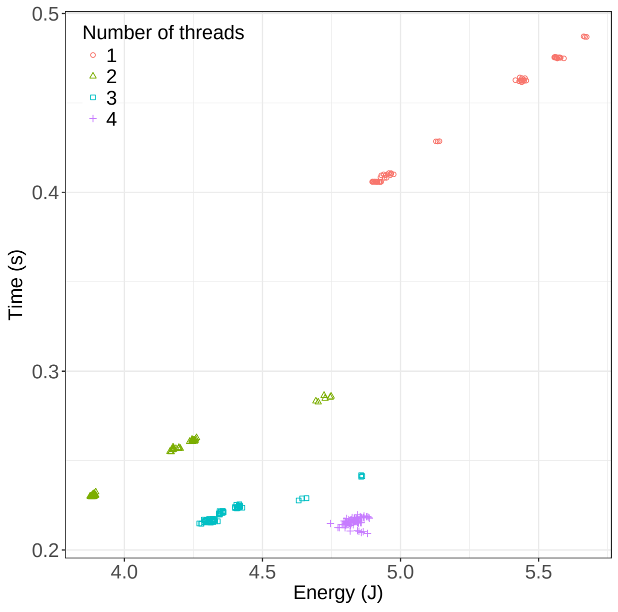

Figure 12. Intel IvyBridge i7-3720QM - Execution times in seconds (ordinate) of the Lagrange 1D kernel by energy (in Joules) consumed by the package (abscissa)

```
optimizer.optimize { |options|
  k = generate_kernel(options)
  stats = []
  nbrepeat.times {
    r = k.run(a,b,c)
    e = r[:energy]
    e = e["0.package-0".to_sym]
    e += e["0.dram".to_sym]
    stats.push e * r[:duration]
  }
  best = stats.sort.first
  best
}
```

Listing 5. BOAST code to optimize for energy delay product

information can be used in conjunction with execution time or GFlops/s to decide which kernel version should be used. For instance, instead of using only execution time one may select the most efficient kernel according to energy-delay-product, as shown in Listing 5. Each point in Figure 12 shows the average execution time and energy consumption for the package (CPU) for a combination of optimization parameters: vector pragmas, nontemporal vector pragmas, loop unrolling and number of threads. Details of the kernel used in this study are shown in Listing 6 (page 177). It performs one-dimensional interpolations with Lagrange polynomials of order 3. One could opt to save energy by using two threads, represented by green triangles, or minimize execution time with four threads, represented by the pink crosses.

The energy counters of the CPU are updated once several milliseconds. To measure the energy consumed by kernels whose execution time is less than a millisecond BOAST can repeat the kernel execution several times, measuring the cost of not one but several execution in succession. Other possibility is to increase the size of the problem solved by the kernel, however that may be undesirable as it may change for instance the cache miss ratio.

4.5. **Conclusion on BOAST use**

The BOAST tool allowed us to improve the performance of one main kernel of the GYSELA code. The portability of performance is effective on all the machines we have evaluated. Nevertheless, the code readability is impacted, the BOAST meta-program is quite complex compared to the original Fortran code. We have found a way to circumvent this issue in insulating the difficulties to a kernel generation mechanism well separated from the usual compilation process, and with a separate code directory. Only a HPC specialist is able to intervene on this part, but it does not bother usual developers and users that still have access to equivalent Fortran code.

Main works that have been carried out to obtain this BOAST kernel are: 1) identify what are the parameters for the auto-tuning, 2) learn BOAST and write a BOAST program that generate the kernel and scan parameters, 3) port and benchmark the meta-program on 4 different machines, 4) integrate into GYSELA the best auto-tuned kernel while keeping a code base that is readable for standard developers and users. The working time invested for the auto-tuned kernel is quite reasonable, but we will see, over time, if the maintenance of such a kernel is costly or not.

Some unknowns still remain: at which level should be set the kernel in the code and what size the kernel should have ? One can make it bigger by incorporating additional code from the application, one can run the auto-tuned process with much more realistic context –examining a large set of application input parameters that have an impact on performance–, one can introduce hardware and threading effects (NUMA memory access, threading deployment ...). Each new element will increase complexity of the meta-program and will enlarge auto-tuning processing time, but can ultimately improve performance a bit more.

5. Conclusion

This paper has studied two complementary auto-tuning approaches to improve the (r, θ) advection of the GYSELA code in terms of performance and portability of performance. On the one hand, we have evaluated a task-based programming approach that enables one to rely on the schedulers provided by StarPU to make execution choices at runtime. We have shown that this approach makes it possible to improve the performance of the code by up to 28%. To achieve this upgrade, we took special care to avoid overheads due to the task granularity, due to the extra memory transfer and allocation associated with tasks, and due to tasks submission. Also, the dynamic aspect of the approach means that the granularity at which it is applied has to remain coarse enough that the overhead of the scheduler does not dominate the gains it provides. This preliminary work is a first step that should prove to be more and more useful with the increased heterogeneity of hardware to come both with accelerators such as GPUs and with imbalance due to the thermal throttling of CPU cores for example. A future work that seems promising would be to describe the task graph at a finer grain in terms of data parallelism so that more threads can work on a given piece of data and thus improve the cache efficiency. Such an achievement requires a task-based description of the algorithm since some tasks can not be parallelized at a finer grain than a 2D plane and efficient use of CPU resources thus requires a parallel execution of fine and coarse grain tasks.

On the other hand, we have evaluated a meta-programming approach that enables one to rely on the BOAST tool to make optimization choices at compile-time. We have identified and studied the impact of five optimization choices in the code. We have shown that the BOAST approach makes it possible to improve the performance of a kernel by up to a factor 1.9 up to 5.7 when compared to the initial code. We have also seen that the optimization choices to make depend on the hardware used and that the portability of performance thus depend on the presence of a machine specific tuning process. The combinatorial tuning process requires a time proportional to the exponential of the number of parameters and is therefore suited to kernels of a few hundred lines of codes with a few tens of parameters maximum. The tuning of more complex optimization spaces is a research field in itself. BOAST proposes a genetic algorithm to solve those problems, but in the case of GYSELA the optimization time were reasonable (less than half an hour for each architecture) and a brute force strategy was adopted. In the short term, a first step to further leverage this work would be to better integrate the optimization process in the GYSELA compilation chain. In the slightly longer term, it would be interesting to identify more optimization parameters to even improve the optimization, especially when it comes to vectorization on Intel Xeon Phi.

It is also interesting to note that the kernel optimized by BOAST corresponds to one of the tasks used in the StarPU approach which demonstrates their complementarity: static tuning at fine grain where the overhead of a runtime would be problematic and dynamic tuning taking information only available at runtime into account when possible. It would however be interesting to study a deeper combination of both approaches. StarPU is able to support multiple implementations of a single task in order to detect and choose the best one at runtime. This could for example be used to choose between multiple versions of a kernel generated by BOAST in order to take into account effects that only appear at run-time such as the concurrence with other threads of the applications.

References

[1] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. Harnessing clusters of hybrid nodes with a sequential task-based programming model. In *8th International Workshop on Parallel Matrix Algorithms and Applications*, July 2014.

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 2011.

[3] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *International Conf. on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012.

[4] Siegfried Benkner, Sabri Pllana, Jesper Larsson Träff, Philippas Tsigas, Andrew Richards, Raymond Namyst, Beverly Bachmayer, Christoph Kessler, David Moloner, and Peter Sanders. The PEPPHER approach to programmability and performance portability for heterogeneous many-core architectures. In *ParCo*. IOS press, 2011.

[5] Julien Bigot, Virginie Grandgirard, Guillaume Latu, Chantal Passeron, Fabien Rozar, and Olivier Thomine. Scaling gysela code beyond 32k-cores on Blue Gene/Q. In *ESAIM: PROCEEDINGS*, volume CEMRACS 2012 of *43*, pages 117–135, Luminy, France, July 2012.

[6] Julien Bigot, Guillaume Latu, Thomas Cartier-Michaud, Virginie Grandgirard, Chantal Passeron, and Fabien Rozar. An approach to increase reliability of hpc simulation, application to the GYSELA5D Code. In Martin Campos Pinto and Frédérique Charles, editors, *ESAIM: Proceedings and Surveys*, number 53 in CEMRACS 2014 – Numerical Modeling of Plasmas, pages 191–210, CIRM – Centre International de Rencontres Mathématiques, Marseille, France, July 2014. EDP Sciences.

[7] Javier Bueno, J. Planas, Alejandro Duran, Xavier Martorell, Eduard Ayguadé, Rosa M. Badia, and Jesús Labarta. Productive programming of gpu clusters with ompss. In *International Parallel and Distributed Processing Symposium*. IEEE, 2012.

[8] Melvin E. Conway. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS'63, pages 139–146. ACM, 1963.

[9] Ludovic Courtès. C Language Extensions for Hybrid CPU/GPU Programming with StarPU. Research Report RR-8278, INRIA, April 2013.

[10] N. Crouseilles, G. Latu, and A. Sonnendrücker. A parallel Vlasov solver based on local cubic spline interpolation on patches. *Journal of Computational Physics*, 228:1429–1446, 2009.

[11] Thierry Gautier, Joao Vicente Ferreira Lima, Nicolas Maillard, and Bruno Raffin. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *International Symposium on Parallel and Distributed Processing*. IEEE, 2013.

[12] V. Grandgirard, J. Abiteboul, J. Bigot, T. Cartier-Michaud, N. Crouseilles, G. Dif-Pradalier, Ch. Ehrlacher, D. Esteve, X. Garbet, Ph. Ghendrih, G. Latu, M. Mehrenberger, C. Norscini, Ch. Passeron, F. Rozar, Y. Sarazin, E. Sonnendrücker, A. Strugarek, and D. Zarzoso. A 5d gyrokinetic full- global semi-lagrangian code for flux-driven ion turbulence simulations. *Computer Physics Communications*, 207:35 – 68, 2016. http://dx.doi.org/10.1016/j.cpc.2016.05.007.

[13] G. Latu, N. Crouseilles, V. Grandgirard, and E. Sonnendrücker. Gyrokinetic Semi-Lagrangian parallel simulation using a hybrid OpenMP/MPI programming. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 356–364. Springer, 2007.

[14] OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 4.5, 2015.

[15] Marc Sergent, David Goudin, Samuel Thibault, and Olivier Aumage. Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System. In *21st International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Chicago, United States, May 2016.

[16] E. Sonnendrücker, J. Roche, P. Bertrand, and A. Ghizzo. The Semi-Lagrangian method for the numerical resolution of the Vlasov equation. *Journal of Computational Physics*, 149(2):201 – 220, 1999.

[17] Top500 supercomputer site. http://www.top500.org. Accessed: 2015-12-01.

[18] Brice Videau, Vania Marangozova-Martin, and Johan Cronsioe. BOAST: Bringing Optimization through Automatic Source-to-Source Tranformations. In *Proceedings of the 7th International Symposium on Embedded Multicore/Manycore System-on-Chip (MCSoC)*, Tokyo, Japan, 2013. IEEE Computer Society.

[19] Brice Videau, Kevin Pouget, Luigi Genovese, Thierry Deutsch, Dimitri Komatitsch, Frédéric Desprez, and Jean-François Méhaut. BOAST: a Metaprogramming Framework to Produce Portable and Efficient Computing Kernels for HPC Applications. *The International Journal of High Performance Computing Applications*, 2017. (To appear).

[20] Wei Wu, Aurélien Bouteiller, George Bosilca, Mathieu Faverge, and Jack Dongarra. Hierarchical dag scheduling for hybrid distributed systems. In *International Parallel and Distributed Processing Symposium*. IEEE, 2015.

A. BOAST tutorial

This subsection details the process of auto-tuning a computing kernel with BOAST. We start with an implementation of a simplified Lagrange 1D GYSELA kernel. We will select the tuning parameters, write the BOAST code which generates and executes the several different possible implementations, and analyse how the implementations affect performance on different platforms.

The first step is to install BOAST and the dependencies of the kernel such as compilers and libraries. BOAST can be installed from RubyGems with the following command:

```
gem install --user-install BOAST
```

The code for the Lagrange 1D kernel is in C. It's made of three functions: *fn_ init*, which initializes the input data; *fn_ advec*, which is the kernel; and *fn_ verif*, which verifies the correctness of the output. The source code for this kernel, with the auto-tuning code, is available in the BOAST documentation[6].

The auto-tuning is controlled with a Ruby script, which contains the kernel implementation and its parameters. This script should first load the BOAST framework:

```
require 'BOAST'
include BOAST
```

After loading BOAST, we need a reference implementation, corresponding to the non-tuned kernel, which we will use to compare the auto-tuned version against. For this we create a *generate_ reference* function, which generates the code for initializing the data, computing the kernel and verifying the result. This function starts by telling BOAST to use C:

```
def generate_reference()
  push_env( :lang => C )
```

Next, we include the code of the reference implementation:

```
  macro_ref = %(
    int fn_advec(double *f0, double *f1, geom_t *geom) {
      int ivpar, iphi, itheta, ir;
      double numres, posx;
      double coef0, coef1, coef2, coef3;

      __assume_aligned(f0, 64);
      __assume_aligned(f1, 64);

#pragma omp parallel private(ir,itheta,iphi,ivpar, \
            numres,coef0,coef1,coef2,coef3,posx)
#pragma omp for schedule(static) collapse(3)
      for (ivpar=0; ivpar<NVPAR; ivpar++) {
        for (iphi=0; iphi<NPHI; iphi++) {
          for (itheta=0; itheta<NTHETA; itheta++) {

#pragma vector always
#pragma vector nontemporal
            for (ir=0; ir<DECNR; ir++) {
              FVAL(f0,ir-DECNR,itheta,iphi,ivpar) = FVAL(f0,NR-DECNR+ir,itheta,iphi,ivpar);
            }
#pragma vector always
#pragma vector nontemporal
            for (ir=0; ir<DECNR; ir++) {
              FVAL(f0,NR+ir,itheta,iphi,ivpar) = FVAL(f0,ir,itheta,iphi,ivpar);
            }

#pragma vector always
#pragma vector nontemporal
            for (ir=0; ir<NR; ir++) {
              posx = 1.5;

              FVAL(f1,ir,itheta,iphi,ivpar) =
                -1./6. * (posx-1.) * (posx-2.) * (posx-3.) *
                FVAL(f0,ir-1+0,itheta,iphi,ivpar) +
```

[6]https://github.com/Nanosim-LIG/boast/wiki/Lagrange1d

```
                1./2. * (posx)    * (posx-2.) * (posx-3.) *
                FVAL(f0,ir-1+1,itheta,iphi,ivpar) +
                -1./2. * (posx)    * (posx-1.) * (posx-3.) *
                FVAL(f0,ir-1+2,itheta,iphi,ivpar) +
                1./6. * (posx)    * (posx-1.) * (posx-2.) *
                FVAL(f0,ir-1+3,itheta,iphi,ivpar);
            }
          }
        }
      }
      return 0;
    })

  init_ref = %(
    int fn_init(double *f, geom_t *geom) {
      int ivpar, iphi, itheta,ir;
      double rval, thval, pval, vval;
      geom->rtmp = malloc(NR*sizeof(double));
      geom->thetatmp = malloc(NTHETA*sizeof(double));
      geom->phitmp = malloc(NPHI*sizeof(double));
      for (iphi=0; iphi<NPHI; iphi++) geom->phitmp[iphi]=sin( geom->phimin   + iphi  *geom->dphi );
      for (itheta=0; itheta<NTHETA; itheta++) geom->thetatmp[itheta]=sin( geom->thetamin + itheta*geom->
          dtheta + geom->dtheta*.5);
      for (ir=0; ir<NR; ir++) geom->rtmp[ir] = sin( ir*geom->dr );

#pragma omp parallel for schedule(static) private(ir,itheta,iphi,ivpar,rval,thval,pval,vval)
      for (ivpar=0; ivpar<NVPAR; ivpar++) {
        for (iphi=0; iphi<NPHI; iphi++) {
          for (itheta=0; itheta<NTHETA; itheta++) {
#pragma vector always
#pragma ivdep
            for (ir=0; ir<NR; ir++) {
              rval  = geom->rtmp[ir];
              thval = geom->thetatmp[itheta];
              pval  = geom->phitmp[iphi];
              vval  = 1.;
              FVAL(f,ir,itheta,iphi,ivpar) = rval * thval * pval * vval;
            }
          }
        }
      }
      return 0;
    })

  code_verif = %(
    int fn_verif(double *f, geom_t *geom) {
      int ivpar, iphi, itheta, ir;
      int64_t count, modulo;
      double rval, thval, pval, vval;
      double numres, exact;
      modulo = NR*NTHETA*NPHI*NVPAR/10;
      for (iphi=0; iphi<NPHI; iphi++) geom->phitmp[iphi]=sin( geom->phimin   + iphi  *geom->dphi );
      for (itheta=0; itheta<NTHETA; itheta++) geom->thetatmp[itheta]=sin( geom->thetamin + itheta*geom->
          dtheta + geom->dtheta*.5);
      for (ir=0; ir<NR; ir++) geom->rtmp[ir] = sin( ir*geom->dr +.5*geom->dr);

#pragma omp parallel for schedule(static) private(ir,itheta,iphi,ivpar,count,rval,thval,pval,vval, \
            numres,exact)
      for (ivpar=0; ivpar<NVPAR; ivpar++) {
        for (iphi=0; iphi<NPHI; iphi++) {
          for (itheta=0; itheta<NTHETA; itheta++) {
            for (ir=0; ir<NR; ir++) {
              count=ir+NR*(itheta+NTHETA*(iphi+NPHI*ivpar));
              rval  = geom->rtmp[ir];
              thval = geom->thetatmp[itheta];
              pval  = geom->phitmp[iphi];
              vval  = 1.;

              numres = FVAL(f,ir,itheta,iphi,ivpar);
              exact  = rval * thval * pval * vval;
              if (!(fabs(numres-exact) < 1.e-4)) {
#pragma omp critical
                {
                  printf( "ERROR! at %d %d %d %d, %.5e %.5e %.5e\\n",
                          ir,itheta,iphi,ivpar,numres,exact,numres-exact );
                  exit(2);
                }
              }
            }
          }
        }
      }
      return 0;
    })
```

Since the reference implementation is in C instead of using the BOAST language we need to tell BOAST what arguments these functions expect:

```
  f = Real( "f", :dim => [Dim(ALLOCSIZER),Dim(NTHETA),Dim(NPHI),Dim(NVPAR)], :dir => :out)
  f0 = Real( "f0", :dim => [Dim(ALLOCSIZER),Dim(NTHETA),Dim(NPHI),Dim(NVPAR)], :dir => :in)
  f1 = Real( "f1", :dim => [Dim(ALLOCSIZER),Dim(NTHETA),Dim(NPHI),Dim(NVPAR)], :dir => :out)
  geom = CStruct( "geom", :type_name => "geom_t", :members => [
    Real( "thetamin"),
    Real( "dtheta"),
    Real( "phimin"),
    Real( "dphi"),
    Real( "rmin"),
    Real( "dr"),
    Real( "vparmin"),
    Real( "dvpar"),
    Real( "thetatmp", :dim => [Dim(NTHETA)]),
    Real( "phitmp", :dim => [Dim(NPHI)]),
    Real( "rtmp", :dim => [Dim(NR)])
  ],:dim => [Dim(1)])
```

Now that we have defined the code and arguments for the functions we can import them in BOAST. In the code below, *Procedure* defines the name, arguments and return type for each of the kernels, and *get_ output.print* is used to write the code to the kernel.

```
  p_init = Procedure("fn_init",[f,geom], :return => Int("a"))
  k_init = CKernel::new
  k_init.procedure = p_init
  get_output.print init_ref

  p = Procedure("fn_advec",[f0,f1,geom], :return => Int("a"))
  k = CKernel::new
  k.procedure = p
  get_output.print code_ref

  p_verif = Procedure("fn_verif",[f,geom], :return => Int("a"))
  k_verif = CKernel::new
  k_verif.procedure = p_verif
  get_output.print code_verif

  pop_env(:lang)
  return [k_init, k, k_verif]
end
```

Now that the reference code is finished we need a function to generate the different kernel versions. This functions receives a parameter *options* which defines the values of the following optimization parameters:

vector_pragmas: Enables or disables the use of #pragma vector
omp_num_threads: Number of threads
nontemporal: The use or not of #pragma vector nontemporal
unroll: Level of loop unrolling

```
def generate_fn_advec(options = {})
  default_options = { :vector_pragmas => false, :omp_num_threads => nil, :nontemporal => true, :unroll =>
      false }
  options = default_options.merge( options )
```

Like in the reference code, we need to declare the parameters of the function:

```
f0 = Real( "f0", :dim => [Dim(-DECNR,ALLOCSIZER-DECNR-1),Dim(NTHETA),Dim(NPHI),Dim(NVPAR)], :dir => :
    inout, :align => 64)
f1 = Real( "f1", :dim => [Dim(-DECNR,ALLOCSIZER-DECNR-1),Dim(NTHETA),Dim(NPHI),Dim(NVPAR)], :dir => :out,
     :align => 64)
geom = CStruct( "geom", :type_name => "geom_t", :members => [
  Real( "thetamin"),
  Real( "dtheta"),
  Real( "phimin"),
  Real( "dphi"),
  Real( "rmin"),
  Real( "dr"),
  Real( "vparmin"),
  Real( "dvpar"),
  Real( "thetatmp", :dim => [Dim(NTHETA)]),
  Real( "phitmp", :dim => [Dim(NPHI)]),
  Real( "rtmp", :dim => [Dim(NR)])
],:dim => [Dim(1)])
```

Now we create a function to use the right pragmas according to optimization parameters *vector_pragma* and *nontemporal*:

```
vector_pragmas = lambda {
  if options[:vector_pragmas] then
    pr Pragma(:vector, "always")
    pr Pragma(:vector, "nontemporal") if options[:nontemporal]
  end
}
```

Now we need to specify the BOAST code for this function:

```
p = Procedure("fn_advec",[f0,f1,geom], :return => Int("a")) {
  ivpar = Int(:ivpar)
  iphi = Int(:iphi)
  itheta = Int(:itheta)
  ir = Int(:ir)
  decl ivpar, iphi, itheta, ir
  numres = Real(:numres)
  posx = Real(:posx)
  decl numres, posx
  coefs = (0..3).collect { |i|
    Real("coef#{i}")
  }
  decl *coefs
  pr OpenMP::Parallel( :private => [ir,itheta,iphi,ivpar,numres,posx]+coefs, :num_threads => options[:
      omp_num_threads] ) {
  pr OpenMP::For( :schedule => "static", :collapse => 3 )
  pr For(ivpar, 0, NVPAR-1) {
    pr For(iphi, 0, NPHI-1) {
      pr For(itheta, 0, NTHETA-1) {
        for1 = For(ir, 0, DECNR-1) {
          pr f0[ir-DECNR,itheta,iphi,ivpar] === f0[ir+NR-DECNR,itheta,iphi,ivpar]
        }
        for2 = For(ir, 0, DECNR-1) {
          pr f0[ir+NR,itheta,iphi,ivpar] === f0[ir,itheta,iphi,ivpar]
        }
        if options[:unroll] then
          pr for1.unroll
          pr for2.unroll
        else
          vector_pragmas.call
          pr for1
          vector_pragmas.call
          pr for2
        end
        vector_pragmas.call
        pr For(ir, 0, NR-1) {
          pr posx === 1.5
          pr f1[ir,itheta,iphi,ivpar] ===
            (-1.0/6.0).to_var * (posx-1.0) * (posx-2.0) * (posx-3.0) * f0[ir-1+0,itheta,iphi,ivpar] +
            (1.0/2.0).to_var  * (posx)     * (posx-2.0) * (posx-3.0) * f0[ir-1+1,itheta,iphi,ivpar] +
            (-1.0/2.0).to_var * (posx)     * (posx-1.0) * (posx-3.0) * f0[ir-1+2,itheta,iphi,ivpar] +
            (1.0/6.0).to_var  * (posx)     * (posx-1.0) * (posx-2.0) * f0[ir-1+3,itheta,iphi,ivpar]
        }
      }
    }
  }
  }
  pr Int(:a) === 0
}
```

Listing 6. "Lagrange1D kernel making use of different optimization parameters"

Lastly, we pass the code to a new kernel and return it:

```
  k = CKernel::new
  k.procedure = p
  pr p
  return k
end
```

After both reference code and the kernel generator are completed we can begin using the code:

```
k_init, k_ref, k_verif = generate_reference
k_init.build(:OPENMP => true)
k_verif.build(:OPENMP => true)
k_ref.build(:OPENMP => true)
```

If the reference code builds successfully, we create the input data and verify the reference kernel works:

```
nbrepeat = 32
set_array_start(0)
k_init.run(fn, geo)
stats = []
nbrepeat.times {
  stats.push k_ref.run(fn, fnp1_ref, geo)[:duration]
}
stats = k_verif.run(fnp1_ref, geo)
```

After verifying the result we create an optimization space containing all combinations of parameters:

```
opt_space = OptimizationSpace::new( :vector_pragmas => [ true, false ],
                                    :omp_num_threads => 1..6,
                                    :nontemporal => [true, false],
                                    :unroll => [true, false] )
```

We now give the optimization space to the brute-force optimizer, which iterates over all possible combinations of parameters and runs them out of order:

```
optimizer = BruteForceOptimizer::new(opt_space, :randomize => true)
puts optimizer.optimize { |options|
  k = generate_fn_advec(options)
  k.build(:OPENMP => true)

  stats = []
  nbrepeat.times {
    stats.push k.run(fn, fnp1, geo)[:duration]
  }
```

We compare the result of each iteration with the reference to ensure the correctness of the optimized version:

```
  diff = (fnp1 - fnp1_ref).abs
  puts diff.max
```

Lastly, we output the version which obtained the lowest execution time:

```
  best = stats.sort.first
  puts best
  best
}
```