Journal of Parallel and Distributed Computing 157 (2021) 13–29

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Practical parallelization of scientific applications with OpenMP, OpenACC and MPI

Marco Aldinucci [a,b,*], Valentina Cesare [c], Iacopo Colonnelli [a,b], Alberto Riccardo Martinelli [a,b], Gianluca Mittone [a,b], Barbara Cantalupo [a,b], Carlo Cavazzoni [d], Maurizio Drocco [e]

[a] Computer Science Department, University of Torino, Italy
[b] HPC Key Technologies and Tools national laboratory, CINI, Italy
[c] INAF - Osservatorio Astrofisico di Catania, Italy
[d] Leonardo S.p.A., Italy
[e] T.J. Watson Laboratory, IBM Research, USA

ARTICLE INFO

Article history:
Received 31 October 2020
Received in revised form 28 February 2021
Accepted 31 May 2021
Available online 12 June 2021

Keywords:
Loop parallelism
CUDA
OpenMP
OpenACC
MPI

ABSTRACT

This work aims at distilling a systematic methodology to modernize existing sequential scientific codes with a little re-designing effort, turning an old codebase into *modern* code, i.e., parallel and robust code. We propose a semi-automatic methodology to parallelize scientific applications designed with a purely sequential programming mindset, possibly using global variables, aliasing, random number generators, and stateful functions. We demonstrate that the same methodology works for the parallelization in the shared memory model (via OpenMP), message passing model (via MPI), and General Purpose Computing on GPU model (via OpenACC). The method is demonstrated parallelizing four real-world sequential codes in the domain of physics and material science. The methodology itself has been distilled in collaboration with MSc students of the Parallel Computing course at the University of Torino, that applied it for the first time to the project works that they presented for the final exam of the course. Every year the course hosts some special lectures from industry representatives, who present how they use parallel computing and offer codes to be parallelized.

1. Introduction

The shift toward parallel computing platforms has many drivers that are likely to sustain this trend for several years to come. Software technology is consequently changing: in the long term, writing parallel programs that are efficient, portable, and correct must be no more onerous than writing sequential programs. While re-designing from scratch with an explicitly parallel approach is still the most effective option to achieve scalable and efficient parallel codes, this approach cannot effectively support the industrial adoption of parallel computing key technologies. In this context, human productivity and time-to-solution are equally, if not more, essential aspects than performance. Also, re-design is a dangerous activity because it might impair the code's correctness; typical pitfalls are in numerical stability, validation of results, and random number generators in parallel codes.

In the last three decades, parallel programming methodologies significantly evolved to ease programmers' tasks and improve program efficiency. The common thread of this evolution has been a rise in the level of abstraction of concurrency management primitives. A crucial step in this process has been defining algorithmic paradigms or *skeletons* (as called by M. Cole in the eighties [2,3,6,12,22,30]) for which a pre-defined proper parallel implementation exists. Some of these paradigms have represented real enabling technologies for whole applicative areas. Among them, the *embarrassingly parallel* paradigm (i.e., task farm skeleton) enabling elasticity in cloud computing [35], the *data parallel* paradigm (map and reduce skeletons) enabling the *MapReduce* programming model [27] with its whole array of Apache-BigData solutions, and the *Single-Instruction-Multiple-Threads* (*SIMT*) programming model for GPUs.

These paradigms happen to work well because they are *explicitly parallel* abstractions. The programmers can directly design their applications within a specific programming model and verify the embedded sequential code's compliance with programming model constraints, such as associativity of accumulation operations, con-

* Corresponding author at: Computer Science Department, University of Torino, Italy.
E-mail address: marco.aldinucci@unito.it (M. Aldinucci).

https://doi.org/10.1016/j.jpdc.2021.05.017
0743-7315/

current access to shared data structures, and absence of a persistent state in pure functions. Since the 1970s, many different programming environments based on these concepts have been proposed [9,32]. A well-known example is the Single Instruction Multiple Data (SIMD) paradigm, which expresses data-level parallelism by applying the same operations to multiple data items in parallel through low-level programming constructs (e.g., Intel SSE/AVX ISA extensions [37]). Many other approaches remained research prototypes, but their essence, which consists of promoting to a first-class concept the replication of sequential functions organized in a parametric schema, has eventually become mainstream.

High-level parallel libraries allow programmers to directly define parallel constructs in the form of either higher-order functions (as in Intel TBB [23], Apache Spark [43]) or directives (as in OpenMP [48] and OpenACC [47]). Such constructs can be associated with *semantically meaningful points of the code*, such as function calls and loops [24]. In modern languages, such as C++11, the role of *directives* can also be played by *attributes*, i.e., first-class language statements that allow the programmer to specify additional information. Such information can be used to enforce compilation constraints or specific code generation, including parallelization [25].

A crucial aspect of generative approaches' success has been focusing on loops as meaningful points of the code. This is particularly true for the shared memory model, where the parallel implementation of loops can be expressed as a composition of parallel constructs without the need of partitioning the data structures [7]. Loops are typically used to navigate arrays and to discretize dimensions (e.g., time and random walks). Since several tools to parallelize a single loop exist, a cost-effective method to enhance the performance and robustness of an entire scientific application revolves around three main tasks: 1) selecting which loops we can correctly parallelize, avoiding both to restrict parallelism needlessly and to require too complex code transformations that might affect numerical stability; 2) determining which loops are worth to be parallelized; 3) selecting points in which the data structures are globally consistent for checkpointing, typically at the beginning of a non-parallelizable loop not nested within a parallel loop.

1.1. Education principle: learning-by-doing parallel design with a Montessorian spirit

This work directly aims to define a strategy to teach students how to cost-effectively parallelize real scientific applications, turning it into a teaching tutorial to be distributed in addition to the text book. We believe that, besides students, this approach might be useful also for domain expert practitioners to tune their scientific applications.

The strategy consists of a systematic methodology that follows well-defined steps, working both in the shared memory model for multi-core and GPUs (for example, with OpenMP and OpenACC) and in the message passing model (for instance, with MPI). We support a unifying approach for prompting students to experience *abstraction* as computer science's distinctive tool against other scientific domains. This methodology has important aftermaths. Firstly, students learn how to approach common cases. In this way, they do not get lost in the plethora of corner cases and parallelization tricks that, in most cases, lead to non-portable code (the Internet is full of them). Secondly, students directly experience how to go (by abstraction) beyond the traditional approach of leveraging low-level and platform-specific functionalities. We believe that the low-level programming approach, which could still be justified today in a few extreme-scale applications, has lost all reason for existing for the vast majority of scientific (and non-scientific) codes. And it is a showstopper for industrial adoption.

This approach matured in the last ten years within the course *Parallel and Distributed Computing Systems*, taught by Prof. Marco Aldinucci at the University of Torino. The course program is designed to be a parallel programming primer, and it is structured in four parts. The first three parts cover three low-level parallel programming models: message passing (exemplified with MPI), shared memory (exemplified with pthreads and OpenMP) and GPU programming (exemplified with CUDA and OpenCL). The last part abstracts these paradigms in higher-level approaches to parallel programming: parallelization methodologies (e.g., loop parallelization), skeletons and patterns (data- and stream-parallelism), and programming models (e.g., MapReduce). In addition, from two to four guest industrial researchers are invited each year for special lessons with a twofold goal: bringing to class real-world evidences of industrial applications and facilitating possible internships (e.g., IBM Research, E4 Engineering, Leonardo Company, ENI).

The course adopts two main education principles: 1) Learn-by-doing,[1] and 2) Student choice of activity plus the discovery model.[2] The course adheres to the learn-by-doing principle with a periodic release of programming exercises (home works), followed by solutions discussed during hands-on sessions. Moreover, the second principle is implemented by structuring the exam around project work, namely an algorithm or application to parallelize. Each student (or group of students) can propose an algorithm or a whole application of interest as project work, choosing one or more parallelization approaches and comparing them. According to the second principle, applications in various domains and different complexity are acceptable (after a discussion with the teacher).

The teacher stimulates the students to join in small groups to address complex applications and proposes standard parallelization exercises to students with no own proposals. Students are encouraged to start the project work in the middle of the term to stimulate the *discovery model*. The project activity typically last few weeks. Students learn concepts from working together with teachers using parallel computing tools, rather than by direct instruction (advocated by both Montessori and Dewey). The final exam consists of both the parallelization activity, accompanied by a report explaining the design choices and the achieved performances, and an oral discussion, covering the project work and the general theory learned during the course. The final grade is given by balancing the complexity of the project work, the quality of the report and the oral exam.

The teacher embraces the Montessorian principle that each student proceeds at its speed and along its trajectory, withholding the temptation to make all student marching at the same pace. According to this principle, the absolute performance achieved by the parallel code developed in the limited time of the project work is not a primary evaluation criterion. Instead, the acquired ability to orientate oneself in the different design choices and analyze (ex-post) performance bottlenecks is highly appreciated.

1.2. Outline of the work

After an outline of related works (Sec. 2), Sec. 3 introduces the mentioned methodology. A preliminary version of the same methodology, targeting OpenMP only, appeared in a previous work [19]. In the present manuscript, we substantially extended it by covering all the mainstream parallel programming models: shared memory multithreading for multi-cores, message passing for clusters, and *SIMT* data parallelism for GPUs. In Sec. 4, the

[1] A theory of education expounded by John Dewey, an American philosopher, psychologist, and educational reformer https://en.wikipedia.org/wiki/John_Dewey.

[2] A method of education expounded by Maria Montessori, an Italian physician and educator best known for the philosophy of education that bears her name, and her writing on scientific pedagogy https://en.wikipedia.org/wiki/Maria_Montessori.

methodology is demonstrated on four real-world scientific applications and validated on state-of-the-art parallel platforms. The four considered codes are: the *DiskMass Survey* code, parallelized with OpenMP (Sec. 4.2), the *Spray-web* code, parallelized for GPUs with OpenACC and CUDA (Sec. 4.3), the *SimpleMD* and *Laplace2D* codes, parallelized with MPI (Sec. 4.4, and 4.5). The *DiskMass Survey* code implements a Monte Carlo Markov Chain (MCMC) method and also appeared in [19], whereas the other three applications are novel. The *Spray-web* code is a community codebase with closed source branches. It is industrially adopted to evaluate the impact of new emission sources like roads, construction sites and industrial plants, by simulating particle movement in a vector field. The *SimpleMD* code implements a Molecular Dynamics simulation, while the *Laplace2D* is an elliptic Partial Differential Equation solver. They represent two broad classes of scientific codes requiring a stepwise global and local system knowledge, respectively. Although the methodology aims to guide the design of a vanilla parallel version, the Spray-web application achieves a 3.7× speedup (NVidia T4 vs Intel i7-7700HQ). SimpleMD has been tested on BSC MareNostrum4 supercomputer achieving a maximum speedup of ∼110, while Laplace2D has been tested on the CRESCO6 cluster, achieving a maximum speedup of ∼778. Presenting to students these four examples and their scaling tests would be essential to show the effectiveness of this methodology for different parallelization models.

2. Related work

One of the main issues when dealing with loop parallelism is to find a solid strategy for *scheduling the iterations* over a set of parallel workers, which must ensure both performance and sequential equivalence (at least to the extent allowed by the finite numerical precision of floating-point operations). Automatic loop parallelization techniques usually perform a lattice analysis of data dependencies to explore the space of data/code transformations, facilitating an efficient loop iteration scheduling [13,26]. Since the nineties, several algebraic frameworks have emerged in the compiler community; the *polytope model* [14,41] is an early example.

Nowadays, lattice analysis is almost exclusively left to either the compiler or the runtime layer of a higher-level parallelization library, which offers a much more straightforward and user-friendly interface to developers. Usually, such an interface comes in the form of either higher-order functions, as in Intel TBB, or `#pragma` directives, as in OpenMP or OpenACC [47]. This kind of approach minimizes the learning curve for developers with little or no parallel programming experience, while still allowing more expert users to fine-tune their applications by specifying many optional parameters. Therefore, these techniques have been preferred to lower-level alternatives over time, as explicitly parallel approaches provide more scalability at the cost of a less intuitive interface [7].

The massive amount of sequential codebases pushed the parallel computing community to find suitable techniques to provide fully *automatic parallelization* of serial codes [38,46,59]. The need to ensure correctness in all cases significantly limits the practical effectiveness of automatic approaches. Indeed, some dependencies that can be easily removed through a code review process could prevent exploiting the parallelism in full degree. As a result, the performance improvement over the baseline is often modest. Unfortunately, even approaches based on compiler-level code inspection, such as OpenMP, can often perform much below the expectations for similar reasons. Conversely, a minor and straightforward code reorganization can significantly improve overall performance.

Typically, a significant gap exists between the toy examples provided in OpenMP tutorials and real scientific applications with multi-level nested loops. This work precisely aims at filling this gap, providing a practical but generic enough methodology for loop parallelization of typical scientific code. Even though some examples of the application of OpenMP to serial scientific codes can be found in literature [45,51], the discussion is usually focused on the analyzed case, making it difficult for researchers without parallel programming experience to generalize concepts and apply them to a similar but different problem.

Other tools are trying to go beyond the state of the art, looking for several different parallel patterns in the code, rather than merely loops [9]. Tools such as *Parallel Pattern Analyzer Tool* [28] aim to identify some patterns, including the *map* pattern (i.e., loop-independent loop), but also the *pipeline* and *farm* patterns, which are typical of event processing (streaming).

OpenMP, OpenACC, and MPI are programming standards for parallel computing. OpenMP originally targeted only shared memory multi-core platforms and it evolved over time to support offloading loops onto GPUs and other accelerators [48], which is also the main target of OpenACC [47]. MPI is the reference programming interface for message passing distributed memory systems [44]. All these standards are presented in the parallel computing course at the University of Torino as a fundamental part of any parallel programmer's toolbox. They are so well-known that they do not need an extensive description. Comparing their pragmatics helps in learning to evaluate which one fits a specific application. In the domain of distributed shared memory systems, several less mainstream languages and frameworks have been proposed that also support loop parallelism, such as Chapel [18] and X10 [21].

2.1. Can we just bind OpenMP to multi-cores and OpenACC to GPUs?

OpenMP started with multithreading in the (cache-coherent) shared memory model. Although it eventually provided support for other programming paradigms (e.g., parallel regions and tasks), OpenMP is still prominently used with loops [48]. OpenACC (for open accelerators) started with loop parallelization for heterogeneous systems but did not yet reach OpenMP's maturity for multithreading [47]. For many years (and very practically), OpenMP was associated with multi-core targets and OpenACC with GPUs. In both frameworks, the programmer can annotate C, C++, and Fortran source code to identify the areas that should be accelerated using compiler directives and additional functions.

More recently, the scenario has become more blurred. OpenACC 2.6, targeting both multi-cores and GPUs, is fully supported by a mainstream compiler such as gcc10. OpenMP 4.5 provides a substantial improvement in its support for programming accelerators and GPU devices. The target platform is no longer crucial to choose between the two. Nevertheless, differences exist, although they are more subtle.

OpenACC uses directives to tell the compiler where and how to parallelize loops and to manage data between potentially separated host and accelerator memories. The OpenMP approach is a more prescriptive, general-purpose parallel programming model where programmers explicitly spread the execution of loops across a team of threads executing on one or more underlying types of parallel computing hardware. The OpenMP directives instruct the compiler to generate parallel code in a specific way, leaving little to the compiler's discretion and optimizer. The compiler must do as instructed. For instance, an OpenMP parallel `do` or parallel `for` directive does not guarantee that a loop is loop-independent. Instead, it instructs the compiler to schedule the iterations of that loop across the available OpenMP threads according to either a default or user-specified scheduling policy. The programmer shares the responsibility with the compiler about correctness of the generated code, possibly dealing with data races through OpenMP-supplied synchronization constructs. The programmer also retains responsibility over execution aspects such as parallelization and scheduling.

By contrast, an OpenACC parallel loop directive is a higher-level construct, by which the programmer simply mark a code section as loop and relies on the compiler for its realization.

2.2. The durable message passing interface (MPI)

Despite several research attempts [29], a similar mainstream directive-based loop parallelization framework has not yet emerged in the distributed memory model, where the durable Message Passing Interface (MPI) standard [44], with send/receive, broadcast, reduction operators, and global synchronizations (barriers), is still the paradigm of choice to construct parallel applications composed of tens to hundreds of thousands of communicating processes. At the low-level tier of the API, each MPI application is conceived as a single program designed on developer-based precise knowledge of the whole code, partitioning of data, and communication overheads. Notwithstanding, platforms have evolved substantially over the last twenty years. Accordingly, the way of exploiting MPI has also evolved, including more collective operations such as the recently introduced MPI 3.0 neighborhood collectives intended to provide support for sparse collective communications both for general communication graphs (no structural restrictions) and for highly structured graphs like stencils over a cartesian topology where processes are organized into d-dimensional tori or meshes [31,53]. On the same line as neighborhood collectives, other higher-level constructs have been included in MPI, aiming at supporting broad ranges of applications, from scientific cases to deep neural networks.

2.3. Performance metrics

The performance metrics of strong and weak scalability are used to predict and test a parallel code's performance.

Strong scaling represents the ability of a software to solve a problem of fixed size faster with a larger amount of computing resources, and it is strictly related to the notion of *Speedup* of a program. The speedup of a parallel algorithm is defined as $S(N) = t_s/t_p(N)$, i.e., the ratio between the time t_s taken by the best sequential algorithm and the time $t_p(N)$ taken by the parallel algorithm against the number of processing elements N. The ideal speedup is linear. Nevertheless, in a real scenario, it is limited by those portions of the code that cannot be parallelized. More precisely, as Amdahl stated in 1967 [10], there exists an upper bound for a program speedup. Let $p \in [0, 1]$ be the fraction of time spent (by a serial processor) in the part of the code that can benefit from parallelization, and $s = 1 - p$ the fraction of time spent in the serial part of the code. Assuming an ideal speedup of the parallel part, the execution time on N processors is no better than $t_p(N) = s\ t_s + p\ t_s/N$. Amdahl's law says that the maximum speedup is given by $S_{Amdahl}(N) = t_s/t_p(N) = t_s/(s\ t_s + (1 - s)\ t_s/N) = N/(1 + (N - 1)s)$, meaning that the maximum speedup is strongly bound by the sequential fraction of the code s, and $S_{Amdahl}(N) \to 1/s$, when $N \to \infty$. Therefore, in practice, a code with serial fraction $s = 5\%$ is bound to a maximum speedup of 20. Unfortunately, Amdahl's law does not consider all the overheads introduced by a parallel implementation, e.g., communications and synchronizations among different workers or initialization of processes/threads. Indeed, actual performances of a program are usually worse than those derived from such law.

Whereas strong scaling is investigated for a problem of fixed size, *weak scaling* is investigated for one of variable size, keeping constant the amount of work assigned to each computing resource. Gustafson's law, formulated in 1988 [33], proposes to measure the speedup by scaling the problem size to the number of processors instead of fixing the problem size. Defining p, s, and N as before, Gustafson's law does not derive the execution time of the parallel code from the sequential one as Amdahl's law does, but it rather considers to keep constant the time t_k spent in each processor by fixing the problem size solved in each processor. As result, the global problem scales while N increases. The latency of serial execution is $T_s(N) = (s + p)\ N\ t_k$, and the parallel execution is $T_p(N) = s\ t_k + p\ N\ t_k$. The *Scaled speedup* can be written as $S_{scaled}(N) = (s + p\ N)/(s + p) = N + (1 - N)s$, which is a linearly decreasing function against the fraction of the serial code s, and a linearly increasing function against the number of processing elements N. Observe that for a fully parallelizable code, the time spent by a problem of size $O(N)$ to run on N processors will remain constant, but also that the scaled speedup does not have an upper limit and it is not bound by s, which determines the maximum scaled speedup growth rate. Weak scaling is easier to achieve than strong scaling.

3. A cost-effective methodology to parallelize sequential applications

In this section, we detail a methodology to achieve the first version of a parallel fault-tolerant code, starting from sequential scientific codes. The methodology pretends neither to be fully automatic, nor to address the parallelization of algorithms exhibiting an inherently sequential behavior, which are not so frequent and that are not anyway the right testbed for inexpert parallel programmers. The four fundamental points that make this methodology important to transmit to students are: 1) it provides an effective parallelization of the original application, with a limited re-designing effort; 2) it can be applied with different parallelization models; 3) it is systematic, i.e., it is always based on the same four steps that have to be executed in sequence; 4) it provides the programmers with metrics and knobs to play with performance, helping students to understand that performance tuning is complex and requires further training. The course would clearly impart a methodology to parallelize scientific codes that goes beyond the copy-paste from library or language tutorials and code samples.

The methodology we advocate guides non-expert parallel programmers in the parallelization of an entire application, rather than automatically solving a single complex loop. The latter problem is beyond the scope of the present work, and we refer the reader to literature for this aspect (see Sec. 2). In reality, most scientific applications do not rely on complex iteration schema, but rather revolve around several loops, either nested or in sequence. These loops can be distinguished into *for-loops* and *while-loops*, where the former iterate over arrays of data and the latter iterate up to a given convergence criterion [58]. The loop body might exhibit a network of dependencies, among both different loops and other iterations of the same loop. Some typical scientific codes whose control flow statement is made of loops are: 1) *Particle simulations*, where an internal loop computes quantities related to each particle, and an external loop advances the simulation time step; 2) *Optimization algorithms*, where one or more internal loops iterate over a subset of the solution space, and an external loop updates the best solution and the heuristic parameters; 3) *Ordinary differential equations (ODE) solvers*, where internal loops iterate over different functions or subsystems, and an external loop advances the time step.

The proposed parallelization methodology consists of four steps: 1) Identify all parallelizable loops in the code, according to a depth-first search strategy; 2) Evaluate the potential performance gain obtainable by modifying each parallelizable loop, filtering out those that are not worth the parallelization effort; 3) Make each of the remaining candidate loops self-contained to remove true data dependencies among different iterations; 4) Use non-parallelizable loops to implement a checkpointing logic to support stop-resume behavior.

We will detail these steps in the remaining sections. As we discussed in Sec. 2, practical tools such as OpenMP and OpenACC make it trivial to parallelize a loop in the shared-address model (CPU and GPU), requiring programmers to simply add appropriate directives right before the identified loops. In the distributed memory model, where fully automatic loop parallelization is not as mature, we advocate (again) a semi-automatic methodology that categorizes the problem according to the true dependencies patterns. As in Wilkinson & Allen's textbook [57], we distinguish problems in order of complexity in 1) *embarrassingly parallel*, 2) *locally synchronous*, and 3) *globally synchronous*. We will discuss the approach we advocate in Sec. 3.5.

3.1. Identify parallelizable loops

We say that A and B are nested loops when the statements of loop B are a proper subset of the statements of loop A. In the most general case, due to procedure/function calls, the described loop inclusion relationship generates a (cyclic) graph of loops and is too weak to identify parallelizable loops. For this, given a code containing multi-level nested loops, it is useful to induce a partial order relation in the inclusion graph using some other relations to turn the graph into a tree. A good example is the domination relationship [11], which induces the *loop-nest tree*. Each node of this tree refers to a distinct loop, and a node B is a child of a node A if they are nested and no other loop appears between them. To put all loops in the same tree, we can consider the entire program body as a pseudo-loop with only one iteration, and we can use it as the root of the tree.

A generic and formal treatment of this concept requires some technicalities from graph theory. However, in many common cases it is quite simple to construct such a tree just by carefully analyzing the code. Such representation of multi-level nested loops suggests a *depth-first search* approach to parallelization, considering one loop at a time starting from the most external level. In this setting, we consider the single iteration of the loop as an *atomic work unit*, and a synchronization barrier is (implicitly or explicitly) placed at the end of each loop to preserve potential inter-loop dependencies.

Bernstein's seminal paper [15] clearly states that the problem of determining if two arbitrary program sections are parallelizable is undecidable in the general case. Then, it offers sufficient conditions to assert that two sections can be executed in parallel by way of three kinds of data dependencies: *true dependencies*, *anti dependencies*, and *output dependencies*. They lead to categorizing loops as *loop-independent*, when iteration i does not depend on any iteration $j < i$ for each $i < N$, and *loop-carried*, when $\exists n \geq 1$ s.t. iteration i depends on iteration $i - n$.

Loops with independent iterations can be trivially parallelized, whereas things get more involved in the presence of loop-carried dependencies. Unfortunately, loops that are not written with a parallel mindset can sometimes contain unnecessary dependencies. These dependencies are generally due to certain sequential coding habits, such as "reusing" variable names for other purposes (inducing anti and output dependencies). They can be automatically removed using techniques such as *variable privatization*, i.e., using multiple copies of the same variable. Loop induction variables are a typical example of variables that can be privatized. True dependencies are much harder to address and have been the object of intense research [4,26]. Loop-carried dependencies can often be addressed by transforming the loop into some new form, in which dependencies are either removed or arranged to occur at a sufficient distance to avoid conflicts among concurrent iterations. A paradigmatic example is the substitution of an accumulator variable with a *reduce* (higher-order) function (over an array of privatized variables). However, not all true dependencies can be eliminated via a *reduce* function. A typical case is when the accumulation operation is not associative, as it happens when building a sequence of pseudo-random numbers from a stateful function, whose state summarizes the history of generated numbers and cannot easily be parallelized.

A practical way to parallelize a program is to descend the loop hierarchy until a parallelizable loop is found. Unfortunately, following the control can be difficult in the presence of conditional branches, which can make the parallelizability of a loop depending from input data and therefore not statically decidable. In this case, it is useful to reduce complexity by identifying the most computationally demanding paths in the control flow graph and enclosing them in dedicated procedure calls. Another potential source of complication arises in those cases when a function containing a loop in its body is called multiple times in the code. In such a scenario, the same loop can appear multiple times in different positions of the hierarchy, with different parallelizability properties each time. In this case, a good strategy would be to maintain a serial version of the function for the non-parallelizable cases, together with one or more parallel versions for the others.

3.2. Evaluate potential performance gain

Every time the previously described depth-first search encounters a parallelizable loop, it is necessary to evaluate the potential benefit introduced by a parallel implementation. Indeed, parallelizing always comes with a certain amount of overhead, introduced by load imbalance and synchronizations among different workers. When the actual computation time of a work unit (called *grain*) becomes too small, parallelization can result in even worse performances than the original serial version. On a modern multi-core platform, the mainstream frameworks such as OpenMP or Intel TBB exhibit a lower limit for the grain on the order of tens of thousands of clock cycles [24]. Finer grains can be addressed only with lock-free programming frameworks, such as Fastflow, that can support grains down to hundreds of clock cycles [6]. On the other extreme, multicomputers and clusters are naturally subject to network latency (no less than 1 µs) and throughput (in turn inducing additional memory copies). Given that, they need to operate with grains in the range of milliseconds to seconds to gain decent scalability.

Both the *number of iterations* in a loop, which affects the maximum obtainable degree of parallelism, and the *total time* spent by the program inside the loop, which determines the maximum achievable speedup, should be taken into account when planning a loop parallelization. When these estimations are complicated, due to the presence of a high number of branching constructs or external procedure calls, a call-graph tool like Callgrind [56] can be helpful.

In general, it would be better to parallelize an outer loop instead of one of its nested counterparts. Indeed, this strategy minimizes the introduced overheads, e.g., for thread creation and synchronization, even if we cannot define a better strategy. Nevertheless, if a loop has very few iterations, the parallelization of one or more of its inner loops can lead to better results. Furthermore, if the code runs on many processors, the parallelization of both inner and outer loops can be even more convenient [49]. An effective greedy technique would be to start parallelizing the outermost suitable loop, where suitable means both feasible and convenient. Then, the loop hierarchy can be further explored, parallelizing suitable nested loops until either performance requirements are met or no noticeable speedup is brought by further optimizations. Once all candidates for parallelization have been identified, it is worth evaluating the maximum expected performance gain by investigating the potential strong and weak scalability of the parallel code.

3.3. Make loops self-contained

Once we have identified a loop that is worth parallelizing, it is necessary to transform the iterative construct into a self-contained procedure call. If the code targets a many-core accelerator with a local address space, all the externally declared variables referenced inside the loop body must be passed by value to the newly created procedure. Otherwise, only the set of variables accessed by write operations should be passed by value. Some modern programming models for hardware accelerators, such as latest versions of CUDA, provide a unified address space abstraction between host and device memory, managing data transfers under the hood and considerably reducing the programming effort.

Frequently, a loop is used to iterate over an array of inputs to produce an array of outputs. Still, it is not uncommon that such a loop is immediately followed by another loop that combines all the produced elements in a single value, utilizing an associative binary operator (e.g., the sum or the product). When thinking about parallel implementations, this particular pattern can be transformed into a *reduce* operation. Given N workers and an input array of $k \gg N$ elements, a reduce pattern can produce the final output in $O(k/N + \log_2 N)$ time steps.

Both the transformation of iterative constructs into self-contained procedure calls and the implementation of the reduce function can either be performed manually or left to an external library like OpenMP or OpenACC. It is always recommended to start with the latter approach, since it is much easier and faster to implement and can guarantee better performance portability among different hardware architectures. Then, we can resort to a manual implementation when necessary.

Random number generators A common problem related to making the loops self-contained revolves around eliminating the interaction via global variables induces by stateful functions, such as system-level or global Pseudo-Random Number Generator (PRNG). Stochasticity is a critical component of many diverse scientific applications, ranging from Bayesian predictive models to Monte Carlo-based simulations of complex systems. In general, all these approaches introduce randomness in their logic by sampling values from a given probability distribution, which is commonly approximated by a PRNG.

PRNGs are stateful objects that approximate genuinely random numbers with actually deterministic numbers. These numbers can be reproduced if the state of the PRNG is known. PRNGs appearing in sequentially executed portions of a program do not need special care, but their initial state can be captured for checkpointing (see Sec. 3.4). On the contrary, their parallelization requires special care. Firstly, the PRNG implementation should be *thread-safe*, i.e., referenceable concurrently from multiple threads without side effects, and *reentrant*, i.e., always returning the same results when called with the same input arguments. Secondly, to enforce reproducibility, the random sequence generated in each parallel section should be *deterministic*, thus independent of their relative execution order. This goal is achieved by privatizing the random induction variable. In object-oriented languages, this can be easily achieved by using an array of PRNG objects, one for each concurrent work unit. Thirdly, to enforce correctness and reproducibility, the array of PRNG objects should be initialized with a seed generated with a master PRNG. Moreover, such PRNG should be implemented using a different algorithm, as using the same algorithm reduces the period and induces loss of distribution uniformity in generated numbers. Once the master PRNG seed is fixed, the sequence of random numbers generated in each parallel section should be deterministic. Fourthly, programmers should know that parallelizing a section of code with PRNG breaks *sequential equivalence*, i.e., the results computed by the sequential and the parallel codes are different. It is a programmer's duty to ensure that the sequential and parallel codes compute the same stochastic process.

3.4. Implement checkpointing logic

Sequential regions of a parallel code do not provide a gain in performance. Still, they can provide an advantage: since they define a global order in the program's execution, they can be used as checkpoints. A *checkpoint* is a snapshot of the entire state of the process at a given time, containing all the information needed to restart the process from that point [40]. Usually, checkpoints are recorded on a *stable storage*, i.e., a persistent storage with some reliability requirements.

Two essential concepts related to checkpoints are *checkpointing overhead*, i.e., the increase in the total execution time caused by the introduction of the checkpointing procedure, and *checkpointing latency*, i.e., the time needed to save the checkpoint. An appropriate checkpointing strategy aims to minimize the first quantity. In order to do that, two different approaches are possible. The first is to minimize latency, either using more advanced storage and communication technologies or reducing the amount of data that must be stored. The other is to store checkpointing data *asynchronously*, reducing overhead regardless of latency. Often, a combination of the two gives the best results.

For example, a checkpoint can be defined in a random sequence of numbers initialized with a given seed. If the application fail-stops at a given point in the sequence, it is possible to restart it from the same point, provided that the state of the random number generator is saved at every iteration in a persistent storage. We will see in Sec. 4.2 how this procedure can be particularly useful in Monte Carlo Markov Chains (MCMCs), which can be quite computationally expensive.

3.5. Parallelization of loops with MPI

Parallelizing loops in the distributed memory model exhibits two additional factors of complexity with respect to the shared memory model discussed so far. The first, quite trivially, is that the data structures must be distributed (replicated or partitioned) on different processing nodes. Therefore, write operations on data replicas generated in different processing elements should be explicitly re-conciliated with direct communications (either 1-to-1 or collective). The second factor revolves around the (relatively) large latency of inter-node communications, requiring to reduce the frequency of communications by setting up a large-enough grain of concurrent work units. It is worth noting that this aspect also affects the reconciliation process, requiring a careful design of the communication plan.

We advocate the idea of parallelizing loops by mapping them into skeletons [9,12,22,32], i.e., pre-defined parallelization schemas. To parallelize the loops we consider data dependences. We distinguish three cases: embarrassingly parallel, globally synchronous, and locally synchronous.

In an *embarrassingly parallel* case, a loop iterates over independent data, which are not partitioned among nodes. These cases are typically approached by transforming the loop into a *master-worker* (or *farm*) skeleton, i.e., a process schema in which one logical entity distributes data to workers. Each worker then computes a function on its data partition and returns the result to the master. The master-worker approach, when applicable, is the most popular and effective parallelization method [1,12,22]. A plethora of master-worker variants have been proposed in the literature, from the simple centralized master (which is typically the first parallelization exercise in a parallel computing course) to the replicated master schema, to the work-stealing schema where each process acts both as master and worker [50]. Interestingly enough, several

modern BigData frameworks are built on top of a master-worker runtime system [43]. The Disk Mass Survey example, described in Sec. 4.2 as an example for the shared memory model, can be easily re-implemented in a distributed memory fashion as a master-worker, with the reduction operations performed in the master.

The two other cases, i.e., *globally and locally synchronous*, are variants of the data-parallel approach, where data structures are partitioned or replicated across different processes (and possibly compute nodes). Similarly to what has been theorized by Valiant's Bulk Synchronous Parallelism (BSP) [54], the parallel computations often proceed by successive *super-steps*. Each super-step is composed by a computation step, in which each process locally computes a function on a data partition, followed by a communication step, where data partitions are reconciled through inter-process communications (1-to-1 or collective). This communication step might include a synchronization barrier (as in BSP) or a *global* or *local* data exchange. These two latter cases are globally and locally synchronous computations, respectively.

In parallel computing, the data decomposition optimization is one of the crucial design steps for performances. Load balance and communication overhead sensibly depend on data decomposition, and this aspect can hardly be fully automated. However, as understood during the design of the High-Performance Fortran compiler [39], the *Owner Computes Rule* with output data decomposition works typically well in scientific computing. In this approach, each process retains a *partition* of the data to be locally written and a copy of all the input data needed to compute the results (potentially including replicated data).

Sequential loops whose iterations (or ranges of them) admit a partition of write-accessed data can be turned into parallel Single Program Multiple Data (SPMD) processes owning these partitions. Then, these processes can exchange data periodically, either locally or globally depending on data dependency patterns. This approach is quite general but not very scalable. Any computation stencil is admitted, including ones whose extent is potentially all the data (or randomly selected inputs from the whole data). As an example, in Sec. 4.4, we present the parallelization of a Simple Molecular Dynamics application, where several second-level loops are parallelized by partitioning iteration blocks into MPI processes that globally `Allgather` (or `reduce`) all the data produced in the previous super-step. According to the proposed methodology, these loops are the outermost parallelizable loops. Several recent applications of parallel computing, including the *Distributed Stochastic Gradient Descent* algorithm used to train Deep Neural Networks, are globally synchronous [55].

In the case where a loop describes a stencil computation, where each data element depends of its neighbors, the locally synchronous paradigm applies. A practical method to turn a loop (or loop nest) into a locally synchronous computation is to transform the algorithm into a block algorithm, where the data to be locally written are partitioned in tiles and traversed in order. Outer d loops traverse tiles (in d dimensions), inner loops move inside each tile (in d dimensions). This *tiled* sequential algorithm can be transformed into a parallel code by assigning each tile to a process in a d-dimensional cartesian topology. Neighbor processes must then exchange tiles borders at each super-step (halo-swap). This technique fits many scientific computations (e.g., simulations and PDE solvers) and, thanks to the constant degree of communications per process per step, it is typically very scalable. For this approach, the MPI standard (from version 3.0) directly supports neighbor communications over cartesian topologies as a first-class concept. In Sec. 4.5, a Laplace 2D solver is presented as an example, the code itself being a mock-up of the PLUTO code [42].

4. Experimental validation

In this section, we present how we applied the methodology to four applications using different technologies; for each application we show the performance we achieved (strong and weak scaling). The parallelization of *DiskMass Survey* and *Spray Web* applications, for different reasons, clearly exceed the effort a single exam should require. The DiskMass Survey is a novel scientific code and the development of the sequential version has been a non trivial effort (happened before the course). The Spray Web code is very long Fortran code with several numerical stability problems emerged during the testing of the parallel version (solved after the exam). Both applications have been proposed by two students because they was directly interested to the parallelization of these codes. The other two kernels (simpleMD code and the 2D Laplace solver) better represent the expected complexity of the code to parallelize for the exam.

4.1. Execution environments

In this subsection, we present the technical specification of the systems used for our experiments.

OCCAM@UNITO is a modular cluster consisting of FAT nodes (4 Xeon-E7@2.1GHz-12cores 768 GB RAM), LIGHT nodes (2 Xeon-E5@2.5GHz-12cores 128 GB RAM) and GPU nodes (Xeon-E5@2.5GHz-12cores 128 GB RAM 2 NVIDIA-K40).

MareNostrum4@BSC with 3456 nodes, each node consists of 2 Xeon-Platinum8160@2.1GHz-24cores 96 GB RAM and Intel Omni-Path (100 Gb/s) interconnection.

CRESCO6@ENEA has 434 nodes, each consisting of 2 Xeon-Platinum8160@2.1GHz-24cores 192 GB RAM and Intel Omni-Path (100 Gb/s) interconnection.

4.2. DiskMass survey with OpenMP

The *DiskMass Survey* code models the rotation curves and the vertical velocity dispersions from the mass distributions of 30 disk galaxies belonging to the DiskMass Survey [16], exploring the agreement between the models and the measured data with a Bayesian approach. The physics of the problem and the related C++ implementation are discussed in detail in other works [19,20]. In a nutshell, the code (Algorithm 1) implements a MCMC method with a Metropolis-Hastings acceptance criterion, obtaining the value of the random variate $\vec{x}$ at step $t+1$ from its value at the previous step t.

After a first preparatory phase that imports some data from external files and initializes the MCMC (lines 1–2), we define two PRNGs with the same seed (line 3) in order to sample: a real uniform distribution, $U(0,1)$, for the Metropolis-Hastings criterion (line 26), and a multi-variate Gaussian distribution, $N(\mathbf{0},\mathbf{1})$, used to generate the free parameters (line 17). After that, the main loop (lines 6–28) computes T MCMC steps, iterating on the number of galaxies with two distinct inner loops. They (lines 13–16) rely on a solver of the Poisson equation ($\nabla^2\phi = 4\pi G\rho$) to approximate the gravitational potential for each galaxy, which is needed to derive the corresponding χ^2. These χ^2 values are then summed to obtain the global likelihood. At this point, a second combination of free parameters is randomly generated from the previous one (line 17) and the second inner loop (lines 22–25) repeats the χ^2 computations on these new values. Finally, the new combination of free parameters is accepted or rejected according to a Metropolis-Hastings criterion (line 26). In the Poisson equation, ϕ is the galaxy gravitational potential, ρ is the galaxy density, the source of the potential, and G is the universal gravitational constant.

Algorithm 1: Parallel DiskMass Survey with OpenMP.

```
1  data_import(density, kinematic data, grid features)
2  x⃗_t ← init_MCMC()
3  PRNG_1(seed), PRNG_2(seed)
4  U ← U(0, 1)
5  N ← N(0, 1)
6  for t ← 1 to T do
7      if t mod 1000 == 0 then
8          save(PRNG_1, PRNG_2)                          // checkpoint
9      χ²_tot(x⃗_t) ← 0
10     omp_set_dynamic(0)
11     omp_set_num_threads(Nthreads)
12     #pragma omp parallel for shared(χ²_j(x⃗_t)) reduction(+: χ²_tot(x⃗_t))
13     for j ← 0 to N_gal − 1 do
14         potentials[j] ← compute_potential(x⃗_t)
15         χ²[j]← compute_chi2(potentials[j])
16         χ²_tot(x⃗_t) += χ²[j]                          // reduce
17     y⃗ ← x⃗_t + jump × G(PRNG_2)
18     χ²_tot(y⃗) ← 0
19     omp_set_dynamic(0)
20     omp_set_num_threads(Nthreads)
21     #pragma omp parallel for shared(χ²_j(y⃗)) reduction(+: χ²_tot(y⃗))
22     for j ← 0 to N_gal − 1 do
23         potentials[j] ← compute_potential(y⃗)
24         χ²[j]← compute_chi2(potentials[j])
25         χ²_tot(y⃗) += χ²[j]                            // reduce
26     x⃗_t ← MH_acceptance_criterion(χ²_tot(x⃗_t), χ²_tot(y⃗), U(PRNG_1))
27     if t mod 1000 == 0 then
28         save(x⃗_t)                                     // checkpoint
```

Because it involves many galaxies, whose quantities are discretized on grids of 100–150K points each, the sequential version of this code can result in quite demanding computations when T is large, especially because the Poisson solver must run twice per galaxy during each MCMC step. Because of this, we decided to apply the proposed semi-automatic methodology to parallelize its execution using OpenMP.

4.2.1. Semi-automatic parallelization of DiskMass survey

Firstly, we identify the code regions with true data dependencies. The main MCMC loop is a sequential process by definition, as the new combination of free parameters is drawn from the previous one at every step. It cannot be parallelized, but it is a good site for the checkpointing logic. The two innermost loops, which independently compute the χ^2 of every galaxy from its gravitational potential, are good candidates for parallelization, but they are both loop-carried because of two accumulator operators (lines 16 and 25, respectively). Nevertheless, since the sum is an associative binary operation, they can be both turned into a parallelizable loop-independent with a parallel reduction, as described in Sec. 3.1. Therefore, we were able to parallelize these regions of the code with the OpenMP library by using a `#pragma omp parallel for shared(`χ^2`) reduction(+:`χ^2_{tot}`)` directive before each loop (lines 12 and 21).

Eventually, we implemented the checkpointing logic. Every 1000 iterations, the program saves to disk both the values of the random generators `PRNG1` and `PRNG2` (lines 7–8) and the parameters chains (lines 27–28). If, for any reason, the execution is interrupted between the $n \times 1000$ and the $(n+1) \times 1000$ MCMC iterations, before restarting the main loop, we can just import from disk the chains made of the first $n \times 1000$ parameters and the two generators saved at step $n \times 1000$, and resume the MCMC loop from iteration $n \times 1000$, instead of restarting it from scratch.

4.2.2. Performance evaluation of DiskMass survey

Having parallelized only the inner for loops, which iterate on galaxies, the maximum ideal speedup that we can achieve is equal to the number of galaxies in the DiskMass Survey (30 in our experiment). We ran the scalability test on a FAT node of the OCCAM cluster (4×12 cores) on $T = 5$ MCMC steps to operate in reasonable timescales. Furthermore, we analyzed how the distribution of threads onto the 4 sockets affects performances. On Linux systems, a process can be launched using numa control—with the `numactl` command—to spread the computation on different sockets and control how data are stored in different cache levels. In particular, the *interleave all* policy keeps all the cores available in all the sockets, whatever the number of threads, allocating memory on all sockets using a round-robin strategy. Conversely, the *block* policy uses one socket at a time until the number of threads saturates its cores. For comparison, we also ran the experiments on a light OCCAM node (2×12 cores), this time without using numa control.

Strong scaling For the strong scaling experiments, we considered the entire sample of 30 galaxies and measured the CPU time of each MCMC iteration using the `gettimeofday` (µs) function, explicitly increasing the number of threads from 1 to 48 at every run (lines 10–11 and 19–20).

The left panel of Fig. 1 shows the speedup (averaged over the 5 MCMC steps) obtained using the numa control with *block* and *interleave all* policies and with the default policy of the machine on the 48-cores node, where t_s and t_p are the execution times for the sequential and the parallel code, respectively. It is worth noting that the ideal linear law holds more or less from 1 to 4 threads, but there is still quite a good linear trend until 9-10 threads. All measures converge to an asymptotic value around 12, but the convergence speed depends on the scheduling policy. Indeed, when applying the *block* policy, the asymptotic limit is reached more slowly, and the execution is less performant between 10 and 34 threads. It means that, in this setting, we need the resources of nearly the entire machine to obtain the maximum gain in performance, while with the other policies, half the machine is enough.

The left panel of Fig. 2 shows the strong scaling with the machine's default policy on the 24-cores node. Qualitatively, this curve trend is similar to the 48-cores one, but this time the speedup reaches its peak value at ∼8.

Weak scaling As the parallel code does not partition a single galaxy onto different threads, the distribution of the sizes of galaxies in the survey affects load balancing. Since weak scaling should be evaluated on inputs of different sizes (i.e., number of galaxies), to avoid the bias due to different size distributions, we considered a synthetic survey composed of 48 galaxies of the same size generated from the real galaxies of the DiskMass Survey.

In the right panel of Fig. 1, we plot the mean time in seconds of each MCMC iteration with respect to the number of processed galaxies for the default, *interleave all*, and *block* scheduling policies. As for strong scaling, the curves related to the default and the *interleave all* policies are quite similar. Weak scalability is quite well satisfied in these two settings: we only lose ∼3 seconds in performance from 1 to 48 threads and only ∼1 second from 1 to 37. On the other hand, we already lose ∼3 seconds in performance from 1 to 14 threads for the *block* mode, and then the time remains nearly constant, with an increasing standard deviation. Furthermore, this test shows that the default and the *interleave all* policies provide the highest efficiency, meaning that we can obtain the best performance gain by using the entire machine.

The right panel of Fig. 2 shows how the weak scaling in the machine's default policy on the 24-cores node behaves like the corresponding curve on the 48-cores platform between 1 and 37 threads. We can conclude that our parallelized code comes with good weak scalability, since the mean time of a single MCMC iteration remains mostly constant for quite a large amount of threads.

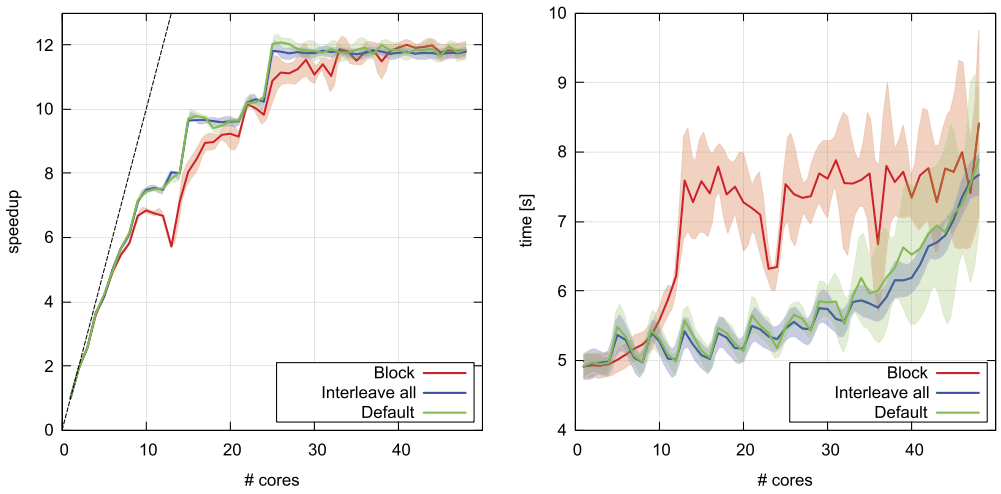


Fig. 1. Left: strong scaling (Amdahl's law). **Right:** weak scaling (Gustafson's law). Both plots refer to a fat node of the OCCAM HPC center (48 cores). The three solid lines represent different scheduling policies to place threads onto cores: numa control in block mode (red line), numa control in interleave all mode (blue line) and the default policy of the machine (green line). (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

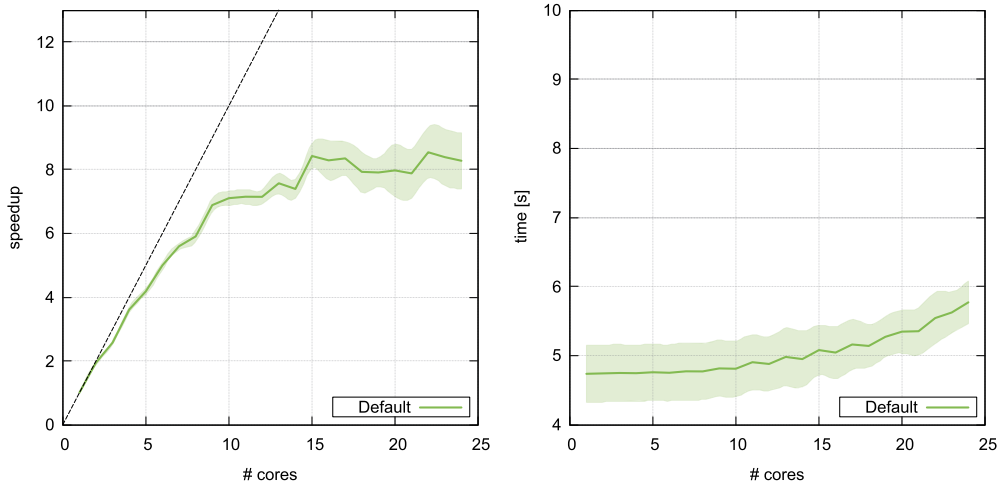


Fig. 2. Left: strong scaling (Amdahl's law). **Right:** weak scaling (Gustafson's law). Both plots refer to a light node of the OCCAM HPC center (24 cores).

4.3. Spray-web with OpenACC (and CUDA)

Spray-web is an application that simulates the dispersion of pollution in a 3D environment. It can take into account spatial and temporal inhomogeneities of both the mean flow and perturbation. The local wind moves the particles of pollution, and the dispersion is the product of velocities obtained as solutions of a system of Lagrangian stochastic differential equations, which reproduces the statistical characteristics of the turbulent flow [17,52].

The Spray-web implementation is a paradigmatic example of old-fashioned engineered scientific code. Developed with a mix of Fortran 77 and 90, it includes functions with hundreds of parameters, very long basic blocks, and several other features that make its automatic parallelization particularly challenging. Moreover, to the best of our knowledge, no GPU-enabled versions of Spray-web were available before this work.

In essence, the code (Algorithm 2) iterates along a discretized timeline of T steps and a collection of `nparticles` particles. At each step of the main loop (lines 3–21), which iterates over time steps, new particles are generated from the existing ones (line 4), and an inner loop (lines 9–15) updates the state of each particle independently from the others. Finally, particles that move out of the simulation domain are removed from the system (line 19).

Given that the number of particles is typically large and the update cost is relatively small, in principle the problem fits well with the GPU/SIMT execution model. The main obstacle for porting, though, is in the absence of modularity in the original implementation. Following the methodology described in Sec. 3, we approach the parallelization of Spray-web using the mainstream loop parallelization tool for GPUs, i.e., OpenACC. For the sake of completeness, we also derive a low-level CUDA parallelization of the code and compare it with the OpenACC approach.

4.3.1. Semi-automatic parallelization of spray-web

For Spray-web, the loop selection is trivial. Indeed, the outer loop cannot be parallelized because it contains true data dependencies (each time step of the simulation after the initial one needs the result of the previous step), but it can be used for checkpointing. Nevertheless, since there were no checkpointing mechanisms in the sequential version, we decided not to implement them in the parallel version to make a fair performance comparison.

Conversely, the inner loop that iterates over particles is the natural candidate for parallelization. Given that, according to the proposed methodology, we firstly transformed the inner loop's body into a self-contained procedure call, and then we parallelized it

Algorithm 2: Parallel grid-stride loop Spray-web (CUDA).

```
1  data_import(sources of particles, nparticles, max num of particles)
2  constant_data_device ← copy_to_gpu(constant_data)
3  for t ← 1 to T do
4      read_only_data ← generate_new_particles(particles_positions,
         particles_velocity, sources)
5      particles_position_device ← copy_to_gpu(particles_position)
6      particles_position_device ← copy_to_gpu(particles_velocity)
7      read_only_data_device ← copy_to_gpu(read_only_data)
8      i ← blockDim.x * (blockIdx.x - 1) + threadIdx.x          // i ← 0 for seq.
9      while i < nparticles do
10         update_position(particles_position_device[i], readonly_data_device,
             constant_data_device)
11         update_velocity(particles_velocity_device[i], readonly_data_device,
             constant_data_device)
12         plume_velocity(particles_position_device[i], readonly_data_device,
             constant_data_device)
13         bounce_service(particles_velocity_device[i],
             particles_position_device[i], readonly_data_device,
             constant_data_device)
14         particles_status_device[i] =
             check_out_of_domain(particles_position_device[i])
15         i ← i + blockDim.x * gridDim.x                       // i ← i+1 for seq.
16     particles_position ← copy_from_gpu(particles_position_device)
17     particles_position ← copy_from_gpu(particles_velocity_device)
18     particles_status ← copy_from_gpu(particles_status_device)
19     remove_out_of_domain(particles_position, particles_velocity,
         particles_status)
20     if t mod 1000 == 0 then
21         save_to_file(particles_position, particles_velocity)
```

Algorithm 3: Parallel grid-stride loop Spray-web (OpenACC).

```
1  data_import(sources of particles, nparticles, max num of particles, nblocks,
     nthreads)
2  $acc data copyin(constant_data)
3  for t ← 1 to T do
4      read_only_data = generate_new_particles(particles_positions,
         particles_velocity, sources)
5      i ← 0
6      !$acc data copyin(readonly_data)
7      !$acc present(constant_data)
8      !$acc copyout(particles_status)
9      !$acc copy(particles_position, particles_velocity)
10     !$acc parallel loop gang vector num gangs(nblocks) vector
         length(nthreads)
11     while i < nparticles do
12         update_position(particles_position[i], readonly_data, constant_data)
13         update_velocity(particles_velocity[i], readonly_data, constant_data)
14         plume_velocity(particles_position[i], readonly_data, constant_data)
15         bounce_service(particles_velocity[i], particles_position[i],
             readonly_data, constant_data)
16         particles_status[i] = check_out_of_domain(particles_position[i])
17         i ← i + 1
18     remove_out_of_domain(particles_position, particles_velocity,
         particles_status)
19     if t mod 1000 == 0 then
20         save_to_file(particles_position, particles_velocity)
```

using both low-level CUDA Fortran and the high-level OpenACC library.

Spray-web with CUDA Fortran The main issue encountered during the parallelization process is due to the Fortran language specification, which does not impose an explicit order of evaluation for expressions. As a result, the *gfortran* compiler used for the serial version and the *PGI Fortran* compiler, which is the only compiler supporting CUDA Fortran kernels, adopt different evaluation orders, strongly affecting the numerical stability between the two versions of the program. It is worth noting that this is not related to parallelization per se: it also happens when compiling the sequential version of Spray-web, and most likely many other scientific codes, with different compilers. The practical way to address this issue is to explicitly impose a unique order by using parentheses in all those cases when numerical stability matters.

After transforming the loop body into a self-contained procedure call, it is possible to convert it into a CUDA kernel by declaring it as `GLOBAL`. Moreover, it is worth noting that, if the CPU and the GPU do not share a common address space (e.g., when working with old versions of CUDA) or if it is necessary to have fine-grained control on data movements to optimize performances, all the data transfers between host and device must be handled explicitly. In this case, a crucial preliminary step is to distinguish, for each kernel, the input dependencies and the output variables. Indeed, the first ones must be copied to the GPU memory before executing the kernel (lines 5–7), while the others must be copied back to the host memory after its termination (lines 16–18). A related issue derives from the use of global variables inside the loop. In the absence of shared address space, such variables must necessarily be passed as additional arguments to the kernel, but finding all these hidden dependencies can be difficult and time-consuming.

The most straightforward strategy to transform a loop in a CUDA kernel is to replace it with a *monolithic kernel*, by substituting all iterations with threads. The loop iterator variable should be privatized and computed from the CUDA block/thread indices pair (line 8), with proper control logic to avoid out-of-bounds accesses.

A more elegant and efficient way to convert loops in kernels is by implementing a *grid-stride loop* [34]. In this case, the loop is not removed, but simply rewritten by initializing its iterator variable to `blockDim.x * (blockIdx.x - 1) + threadIdx.x` (in CUDA Fortran `blockIdx.x` starts from 1) and incrementing it by `blockDim.x * gridDim.x` at each iteration. In this setting, which we adopted in our implementation, each thread can be associated with multiple iterations of the original sequential loop, allowing greater flexibility in performance tuning. Moreover, by incrementing the iterator variable by the grid's dimension at each iteration, it is possible to ensure a maximally coalesced memory access pattern for CUDA threads.

The code is now parallelized, but it can still be slow because it was designed and optimized to run on CPU. In particular, the bottleneck usually comes from the data transfers between host and GPU, which are orders of magnitude slower than normal in-memory copy operations that occur, for example, when passing arguments by value in a function call. In Spray-web, this problem is exacerbated by the absence of modularity in the program design, resulting in subroutines with hundreds of parameters. While in the CPU version almost all parameters can be passed by reference, minimizing the overhead of function calls, when dealing with GPUs all such parameters must be transferred back and forth between host and device memories, with an enormous loss in performances.

The optimal solution would be to rewrite the code in a more modular way, but this would require a huge programming effort. A more realistic option is to search for avoidable data transfers to reduce the overhead, e.g., locate all constant input dependencies of CUDA kernels and transfer them only once at the beginning of the program (line 2). Another way to reduce data transfer overhead is to overlap computations and communications using a combination of memory pinning and CUDA streams. Unfortunately, the optimal implementation of these more advanced implementations is often hardware-specific, falling outside of the concept of the semi-automatic methodology discussed in this paper.

Spray-web with OpenACC For the OpenACC implementation, we firstly added some directives to manage the data transfers between host and device (lines 6–9). As mentioned before, data transfers play a crucial role in GPU codes, and, to obtain decent performances it is often necessary to optimize them carefully. In OpenACC, data transfers can be declaratively managed with the `data`

Table 1
Spray-web experiments (execution time).

	Time (s)		Speedup	
	Input 24H	Input 121H	Input 24H	Input 121H
Seq. Intel i7-7700HQ CPU	166.905	782.532		
CUDA T4 GPU	45.507	227.496	3.67	3.44
OpenACC T4 GPU	44.885	211.922	3.71	3.70

directive. Compilers tend to adopt a conservative approach, copying all the data from the host to the GPU before every kernel execution and transferring them back to the host after each termination. However, the overhead thus introduced can be harmful if a kernel is launched many times. Generally speaking, constant data can safely be left in GPU memory during the entire program execution, read-only data can be copied only before the kernel execution, and write-only data only after kernel termination.

In OpenACC, these optimizations can be explicitly suggested to the compiler with ad hoc data directives, which take a list of variables as arguments. In detail, a `copyin` clause's arguments should only be copied from host to GPU prior to executing the kernel, while arguments of a `copyout` clause should only be transferred to the host memory after the kernel termination. The `copy` clause is used for read-write variables. Another useful clause is `present`, which forces the compiler not to transfer data if they are already stored in memory.

The `parallel loop` directive has be used to parallelize the loop (line 10). The values of `nblocks` and `nthreads` arguments specify the number of blocks and threads per block according to the standard CUDA programming abstraction.

4.3.2. Performance evaluation of spray-web

We tested both the sequential CPU and parallel GPU versions on two distinct input files, resulting in two particle evolution simulations across 24 and 121 hours. For the GPU experiments, we used a Tesla T4 GPU with 2560 CUDA cores, while for the sequential ones, we used an Intel i7-7700HQ CPU. In both cases, the amount of memory was equal to 16 GB.

To fairly compare the two versions, we both fixed the random seed and reset the position of the particles after each iteration. This last operation was necessary to overcome the problem of the evaluation order of mathematical expressions discussed earlier. Execution times and speedups obtained with both the pure CUDA and the OpenACC implementations with respect to the sequential CPU version are reported in Table 1.

It is worth noting that OpenACC results are slightly better than pure CUDA ones, probably due to the limited effort we put in the optimization of the native version. Indeed, while it is true that working directly with explicitly parallel paradigms gives access to much higher flexibility, which translates to a higher potential gain in performance, it is also a fact that reaching that limit requires massive coding efforts from experienced people. Conversely, when the goal is to achieve a good speedup with minimal effort, as in the case of the proposed methodology, higher-order libraries such as OpenACC represent by far the most suitable solution.

4.4. SimpleMD with MPI

The Simple Molecular Dynamics (SimpleMD) application simulates the evolution in position and velocity of a set of particles using the Lennard-Jones potentials. SimpleMD's pseudocode is shown in Algorithm 4. The code takes as input the number of simulation steps (T) and the number of the particles (`nparticles`). For simplicity, we assume `nparticles` to be a multiple of the number `nprocs` of processes allocated for the program execution.

After an initialization phase that *generates* the particles (line 3) and computes the initial forces (line 7), the main simulation loop

Algorithm 4: Parallel SimpleMD (SPMD).

```
   Data: nparticles, number of iterations T
1  rank ← getRank()
2  if rank == 0 then
3      particles ← generate_particles(nparticles)
4  broadcast(particles, 0)
5  nprocs ← getNumberProcesses()
6  nparticles_loc ← nparticles/nprocs        // nparticles_loc ← nparticles for seq.
7  force ← init_forces(particles)
8  for t ← 1 to T do
9      for i ← 0 to nparticles_loc − 1 do
10         k ← rank × nparticles_loc + i                    // k ← i for seq.
11         update_position(particles[k], force[k])
12     Allgather(particles, nparticles_loc)
13     for i ← 0 to nparticles_loc − 1 do
14         k ← rank × nparticles_loc + i                    // k ← i for seq.
15         force[k] ← 0
16         for j ← 0 to nparticles−1, j ≠ k do
17             force[k] ← force[k] + compute_force(particles[k], particles[j])
18         update_velocity(particles[k], force[k])
19     Allgather(particles, nparticles_loc)        // can be optimised using
         "triangular_pattern(particles)"
20     E_p ← 0
21     for i ← 0 to nparticles_loc − 1 do
22         k ← rank × nparticles_loc + i                    // k ← i for seq.
23         for j ← k + 1 to nparticles−1 do
24             E_p ← E_p + potential_difference(particles[k],particles[j])
25     E_k ← 0
26     for i ← 0 to nparticles_loc − 1 do
27         k ← rank × nparticles_loc + i                    // k ← i for seq.
28         E_k ← E_k + kinetic_energy(particles[k])
29     TotalEnergy ← E_p + E_k
30     reduce(TotalEnergy, sum, 0)
```

(lines 8–30) runs for a fixed number T of steps. Each iteration of this loop, in turn, contains four inner loops that:

1. *Update* the position of the particles using the force vectors computed during the last iteration (lines 9–11).
2. *Compute the new force* vectors between each pair of particles using the new positions and update the velocity of each particle using the new forces (lines 13–18).
3. *Compute the potential* energy of the system by summing up the contributions of each particle pair, which depend on the new positions (lines 20–24).
4. *Compute the kinetic energy* of the system by summing up the contributions of each particle, which depend on the new velocities (lines 25–28).

At the end of each iteration, the program computes the system's total energy as the sum of the potential and the kinetic contributions (line 29).

4.4.1. Semi-automatic parallelization of SimpleMD

According to the proposed methodology, we first identify parallelizable loops. The main simulation loop belongs to the loop-carried class because the position of the particles computed at the beginning of each iteration depends on the force vectors computed in the previous step. It could be a perfect candidate to host checkpointing logic. Moreover, neither the first particle-generating routine (line 3) is trivially parallelizable because it imposes a constraint on the minimum distance among a newly generated particle and the existing ones. Conversely, the inner loops in the main one iterate over particles or particle pairs in a loop-independent way, making them suitable targets for our methodology.

In the parallelized version, each process becomes responsible for updating only a subarray of particles with size $\texttt{nparticles}_{\texttt{loc}}$, where $\texttt{nparticles} = \texttt{nparticles}_{\texttt{loc}} \times \texttt{nprocs}$.

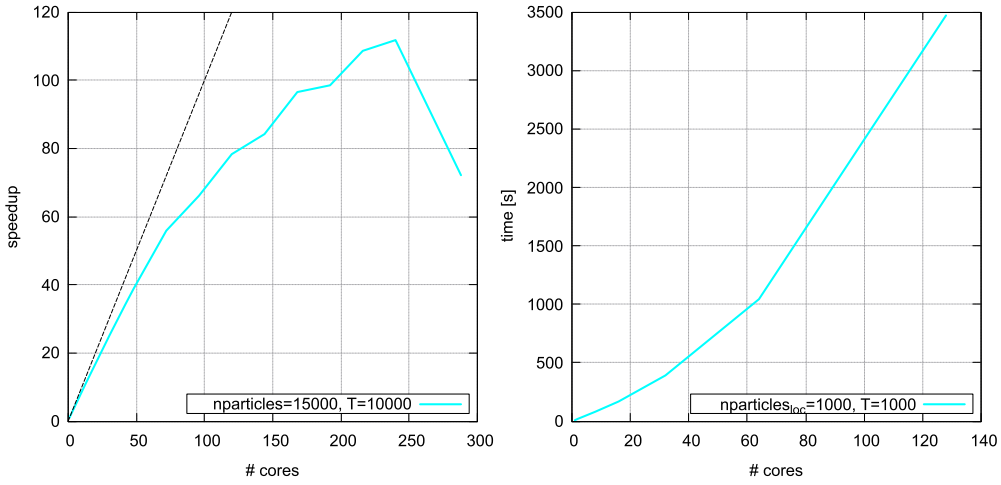


Fig. 3. Left: strong scaling (Amdahl's law). **Right:** weak scaling (Gustafson's law).

In the following paragraphs, we describe the strategies adopted to parallelize the different portions of the code in more detail.

Generation Since a new particle's position depends on the position of all the previous ones, the generation step is left to a single master process. Then, the most natural strategy would be to distribute $\texttt{nparticles}_{\texttt{loc}}$ particles to each process with a scatter primitive. However, since each process needs all the `nparticles` particles to compute forces and potential differences, the array of particles must be broadcasted to each process using the `MPI_Bcast` function (line 4).

Update Now that each process can access the entire particles array, we can focus on a generic iteration of the simulation loop. Positions can be trivially updated without inter-process communications, but then each process needs all the particles' updated positions to compute the new force vectors. After this communication, which is performed by the `MPI_Allgather` function (line 12), each process can independently compute the new forces and velocities for its particles subarray.

Reduction The last step is energy computation. As discussed before, the system's total energy is obtained by summing the kinetic energy of each particle and the potential difference between each pair. In the sequential version of the code, potential differences are computed using a triangular loop, i.e., a nested loop having the same termination condition as the outer one, but whose index is initialized with the index of its outer loop plus 1 (e.g., lines 20 and 23). Such a construct implies that each process needs to receive particles only from the processes with a higher rank and must send its subarray only to the lower-ranked ones, making a standard `MPI_Allgather` communication strongly inefficient. In order to improve the performance, we decided to implement an ad hoc `triangular_pattern` function (line 19). Conversely, the computation of the kinetic energy is trivial since particles can be independently processed one by one. Finally, each process obtains the total energy of its particles subarray and an `MPI_Reduce` function computes the grand total.

4.4.2. Performance evaluation of SimpleMD

For the experimental evaluation, we ran the serial and parallel versions of the code on up to 6 nodes of the BSC MareNostrum4 supercomputer. Results for both strong and weak scaling are reported in Fig. 3.

For the strong scaling problem we set up $\texttt{nparticles} = 15000$ and $T = 10000$. The sequential version took more than 5 hours to complete, while the parallel code with 240 processes finished in less than 3 minutes, achieving a maximum speedup of 111. Scaling to more than 5 nodes resulted instead in an evident performance degradation, meaning that the additional computing power could not compensate for the overhead induced by heavy all-to-all communications (e.g., the `MPI_Allgather`).

Regarding the weak scaling setting, we considered a unit of work $\texttt{nparticles}_{\texttt{loc}} = 1000$ for each process and a number of iterations $T = 1000$, in order to operate within reasonable timescales. The execution time was not constant because some operations, viz., force and potential energy computations, have a quadratic complexity in terms of `nparticles`.

4.5. The 2D Laplace solver and MPI

A 2D Laplace solver (Algorithm 5 left panel) iteratively searches for an approximate solution of the Laplace equation, i.e. a Poisson equation (Sec. 4.2) with a source term ρ equal to 0, on a $N_x \times N_y$ evenly spaced cartesian grid. First of all, the value of the potential ϕ at the boundaries of the domain has to be fixed a priori (line 10), while other points are simply initialized to 0 (line 9). We extend the grid on each side by one point where the Boundary Conditions (BCs) are set (red points in Fig. 4, left panel). Then, for each point in the grid, the new value of ϕ is computed upon the values assumed by all its neighbors in the previous iteration (line 18). The algorithm iterates until ε, the sum of all the differences between the computed potentials in the last two iterations (line 19), goes below a given tolerance *tol* (line 11).

4.5.1. Semi-automatic parallelization of the 2D Laplace solver

The first thing for parallelizing the Laplace solver with the proposed approach is to reorganize the serial code by dividing the global domain in $T_x \times T_y$ equally-sized sub-domains (or tiles). To compute the solution of the equation it is now necessary to firstly iterate over the tiles along each dimension (line 13) and then to iterate over the cells in each tile (lines 14–15), resulting in 4 nested for loops.

The right panel of Algorithm 5 shows in red how it is possible to parallelize with MPI the two external for loops and to combine each sub-solution with the `MPI_Allreduce` function (line 20) to derive the global solution. The decomposition of the domain is performed in parallel through the `MPI_Cart_create` function, which assigns each sub-domain to an individual process according to the topology illustrated in Fig. 4. This function returns a communicator that encodes the new MPI topology. In an execution with `size` processes, the number of processes along each dimension of the grid, `npx` and `npy`, is determined through a manual

Algorithm 5: Seq. and Parallel MPI Laplace.

Seq. Laplace (with tiling)

```
s.1  T_x ← nTiles_x, T_y ← nTiles_y
s.2  n_x ← N_x/T_x, n_y ← N_y/T_y // size of tiles
s.3
s.4
s.5  mallocs: φ0[N_x + 2, N_y + 2], φ1[N_x + 2, N_y + 2]
       // φ0 and φ1 globally describe the problem,
       // boundary conditions need a halo of 2
s.6  φ_A ← φ0; φ_B ← φ1 // duplication of pointers
       // φ_B is step i + 1 (write), φ_A step i (read)
s.7  for i ← 1 to N_x do // init
s.8      for j ← 1 to N_y do
s.9          φ_A[i, j] ← 0

s.10 init_boundary_conditions(φ_A, N_x, N_y)
s.11 while (ε > tol) do
s.12     ε ← 0
s.13     for sx ← 0 to T_x − 1 do // global solver
             for sy ← 0 to T_y − 1 do
s.14             for ii ← 1 to n_x do
s.15                 for jj ← 1 to n_y do
s.16                     i ← sx × n_x + ii
s.17                     j ← sy × n_y + jj
s.18                     φ_B[i, j] ← F(φ_A[i − 1, j], φ_A[i + 1, j], φ_A[i, j − 1], φ_A[i, j + 1])
s.19                     ε ← ε + F(|φ_B[i, j] − φ_A[i, j]|)

s.20
s.21     swap_pointers(φ_A, φ_B)
```

Parallel MPI Laplace

```
p.1  T_x ← nTiles_x = np_x, T_y ← nTiles_y = np_y
p.2  n_x ← N_x/T_x, n_y ← N_y/T_y // size of tiles
p.3  Cartesian_create_comm(COMM_WORLD, 2, np, periods, 0, COMM_CART)
p.4  Cartesian_get_comm(COMM_CART, 2, np, periods, coords)
p.5  mallocs: φ0[n_x + 2, n_y + 2], φ1[n_x + 2, n_y + 2]
       // φ0 and φ1 are partitions of the problem,
       // boundary conditions need a halo of 2
p.6  φ_A ← φ0; φ_B ← φ1 // duplication of pointers
       // φ_B is step i + 1 (write), φ_A step i (read)
p.7  for i ← 1 to n_x do // parallel init
p.8      for j ← 1 to n_y do
p.9          φ_A[i, j] ← 0

p.10 init_boundary_conditions(φ_A, n_x, n_y)
p.11 while (ε > tol) do
p.12     ε_loc ← 0
p.13     Neighbor_alltoall(bounds(φ_A),max(n_x, n_y) + 2,
           bounds(φ_A),max(n_x, n_y) + 2,COMM_CART)
p.14     for ii ← 1 to n_x do // local solver
p.15         for jj ← 1 to n_y do
p.16             i ← ii
p.17             j ← jj
p.18             φ_B[i, j] ← F(φ_A[i − 1, j], φ_A[i + 1, j], φ_A[i, j − 1], φ_A[i, j + 1])
p.19             ε_loc ← ε_loc + F(|φ_B[i, j] − φ_A[i, j]|)

p.20     Allreduce (ε_loc, ε, sum)
p.21     swap_pointers(φ_A, φ_B)
```

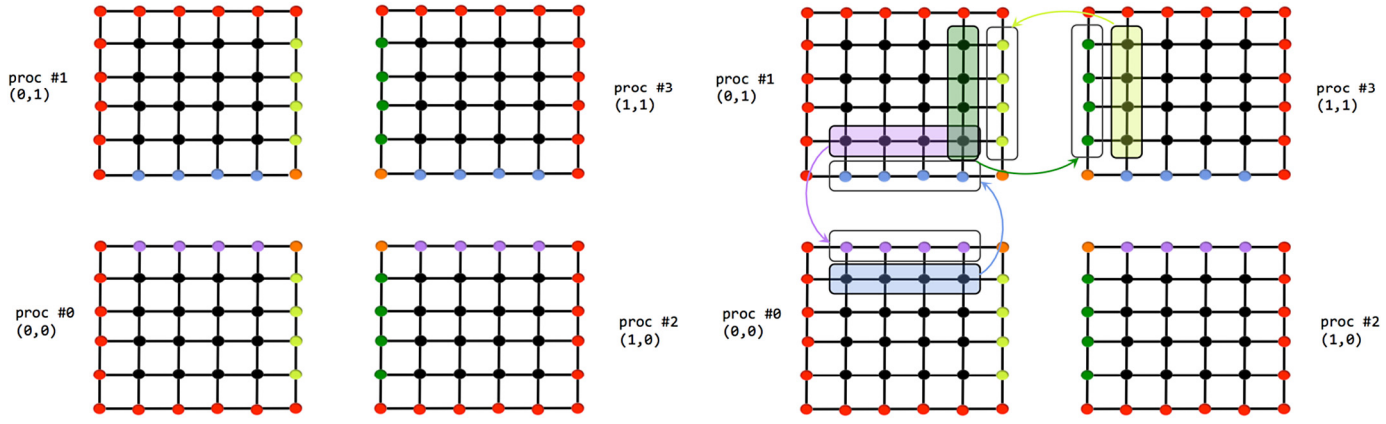


Fig. 4. Left: Example of a Cartesian decomposition between 4 processes of the domain performed with the `MPI_Cart_create` function. Each portion of the global domain is mapped to a single process and to a couple of coordinates. Black dots show the points of the local grids where Laplace equation is computed iteratively. Red dots show the points where physical Boundary Conditions (BCs) are set whereas the dots colored in different ways show the points where inter-processes BCs are defined. **Right:** Halo-swap operation between inter-processes BCs.

maximally squared decomposition, where $\texttt{np}_x = \sqrt{\texttt{size}}$ and $\texttt{np}_y = \texttt{size}/\texttt{np}_x$. $\texttt{np}_x \times \texttt{np}_y$ is equal to `size` when `size` is a square number; if $\texttt{np}_x \times \texttt{np}_y \neq \texttt{size}$ then the domain cannot be decomposed and the execution does not start.

Since we compute the Laplace equation in parallel on different portions of the grid, the number of BCs increases: besides physical BCs set at the borders of the global domain, now inter-processes BCs are also present. The definition of inter-processes BCs requires communication among the different neighbor processes in the new MPI topology, since each process has to send to and to receive from its neighbors the arrays containing the potential computed in the most external interior points of the sub-grid. These arrays are then saved at the boundaries of each local domain such that the Laplace equation can be solved on every process. This halo-swap communication is performed through the `MPI_Neighbor_alltoall` function (line 13), as shown in the right panel of Fig. 4.

4.5.2. Performance evaluation of the 2D Laplace solver

Strong scaling The strong scaling performance of the parallelized Laplace solver has been tested on the CRESCO6 cluster, using an OpenMPI version compiled specifically for Intel processors. We evaluated the solver using a grid of size 32768×32768, which is sufficiently large to guarantee computational times significantly larger than the communication ones, with a sufficiently challenging tolerance of 4×10^{-7}. Moreover, we decided to allocate only half of the cores available on each node (i.e. 24 cores out of 48) since we empirically observed that a greater occupation of the node resources led to a rapid deterioration in performance of the node itself. The left panel of Fig. 5 shows the average speedups on ten executions: Amdahl's law holds until 16 cores but, albeit more slowly, the speedup continues growing along the entire range of considered cores, achieving a maximum value of ∼778 with 2808 cores.

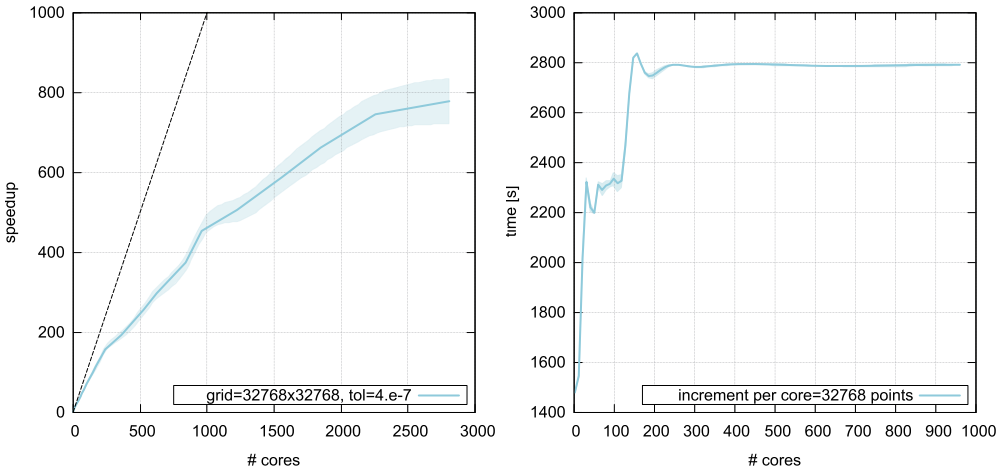


Fig. 5. Left: strong scaling (Amdahl's law) and **right** weak scaling (Gustafson's law) for the Laplace solver code, averaged on ten executions on the CRESCO6 platform. The shaded areas represent the errors on the measurements and the black dashed line is the ideal Amdahl's law.

Weak scaling To investigate the weak scalability of the parallelized Laplace solver, we kept constant the number of grid points per core, meaning that we added 32768×32768 points to the grid for every core used in the run. This implies an increase in the global grid's resolution since its dimensions are fixed; consequently, runs with a different number of processors will converge at different speeds, thus making the results not comparable among each other. For this reason we did not use a fixed tolerance as the ending condition of the program, but we run all the computations for a fixed number of iterations, making the runs comparable. As for the strong scaling, we allocate only half of the cores available on each node. The right panel of Fig. 5 illustrates the weak scaling result, with the times being averaged on 10 executions. The time slowly increases from ∼2200 to ∼2800 s, up to 624 processes, and then it remains constant around this value, showing a rather good weak scalability.

5. Scientific and educational conclusions

This works covers two aspects. First, it presents a novel methodology to devise the first parallel version of a scientific code; the methodology is presented as a semi-automatic procedure (best-practice). Second, it describes how the methodology has been co-designed by the teacher and some of the students of the *Parallel and Distributed Computing Systems* (MFN0795), a course of the M.Sc. degree in Computer Science of the University of Torino, Italy.

5.1. Scientific aspects

In this work, we illustrated a novel methodology that does not concentrate on the parallelization of the single loop, as many methods already presented in the literature do, but of entire scientific applications, designed with a loop-flow structure. This methodology is presented as a tutorial style to become additional material for the course. This approach is particularly suitable for beginners in programming because it both provides a minimal redesign of the sequential version of the application and it achieves this target in a systematic way, always following the same sequence of four steps: 1) identifying all parallelizable loops in the code; 2) evaluating the potential performance gain obtained through this parallelization; 3) transforming these loops in self-contained procedure calls; 4) exploiting non-parallelizable loops to implement a checkpointing logic. Moreover, this methodology can be applied with different parallelization technologies, like OpenMP, OpenACC, CUDA, and MPI, providing a very versatile approach.

In this work we tested the methodology on four different codes: 1) the *DiskMass Survey* code, parallelized with OpenMP; 2) the *Spray Web* code, parallelized with CUDA and OpenACC for GPUs; 3) the *SimpleMD* code, parallelized with MPI; 4) the *Laplace* code, parallelized with MPI. Parallel versions of all these codes have been developed and tested by four students of the course (which are also authors of this manuscript).

The *DiskMass Survey* code illustrates an astrophysical case where we modeled the kinematic profiles of a sample of galaxies considered at the same time with a MCMC. We achieved a speedup of 12, passing from a total execution time of 6.1 Ms (∼71 days) for the sequential version to a total execution time of 0.52 Ms (∼6 days) for the parallel version. We also tested the weak scaling: by increasing the input proportionally to the degree of parallelism, the execution time remains mostly constant.

The *Spray Web* code is a legacy code written with a mix of Fortran 77 and Fortran 90, simulating the dispersion of pollution caused by the local wind in a 3D environment. Such dispersion is modeled as the solution of Lagrangian stochastic differential equations. In this case, we applied our methodology using both a high-level tool (OpenACC) and a low-level tool (CUDA Fortran), addressing some simple optimizations necessary to obtain acceptable results. We tested the sequential and parallel versions with two inputs (a 24 hours and a 121 hours simulations). The OpenACC version led 3.70 and 3.71 speedups, while with CUDA Fortran we achieved 3.67 and 3.44 speedups, respectively.

The *SimpleMD* code simulates the evolution in position and velocity of a set of particles using the Lennard-Jones potentials. We achieved a speedup of 111, passing from a total computing time of ∼5 hours for the sequential version to a total computing time of less than 3 minutes for the parallel version. Regarding the weak scaling, the execution time is not constant because some operations have a quadratic complexity in terms of the number of particles.

The *Laplace* code is a 2D Laplace solver, which solves an elliptic partial differential equation iteratively on a 2D evenly spaced cartesian grid. We used this code because, unlike the previous one, at each iteration the processes communicate only with a subset of neighbor processes. We tested the sequential and parallel versions, achieving a maximum speedup of ∼778 using 2808 cores.

5.2. Educational aspects

The Parallel and Distributed Computing Systems course has been activated by the University of Torino in AY 2010-11 at the

Table 2
History of students' evaluations of the course along the AY when the methodology has been unrolled. The percentage measures the ratio between positive evaluations and all evaluations.

Questions	2016-17	2017-18	2018-19	2019-20
Initial know-how	83%	77%	100%	100%
Load of study	83%	100%	100%	100%
Teaching material	66%	66%	60%	100%
Examination method	83%	89%	100%	100%
Stimulation of interest	66%	89%	100%	100%
Additional activities	66%	57%	100%	100%
Coherency	100%	77%	100%	100%
Teacher availability	100%	88%	100%	100%
Interest in the topic	100%	100%	100%	100%

Physics department, then moved to the Computer Science Department in AY 2016-17. Overall it receives 15-20 students per year from several M.Sc. degrees: Computer Science (90%), Physics (5%), and Mathematics (5%). Prof. Marco Aldinucci is teaching the course from the first edition. The course evolved over the last five years. It started from direct instruction of theory (with coding examples), with an exam consisting in the parallelization of a classic algorithm. Through time, the proposed method, named as *learning-by-doing parallel design (with Montessorian spirit)* in Sec. 1.1, has progressively being unrolled. This method maintains the formal aspects of the course. Still, it aims to differentiate the students in different (possibly overlapping) groups, exhibiting a different attitude to face complex problems. The goal is to offer each group a correctly dimensioned challenge as project work. This process is dynamic, requiring the teacher to work with students during project activity and continuously evaluate the progress of learning. If necessary, the project requirements can be relaxed, e.g., passing from many parallelization methods to a single one or changing the algorithm to be parallelized.

Students have evaluated the course by way of the University of Torino's standard questionnaire (called EduMeter). Each student should mandatorily evaluate the course before the exam (oral discussion). Teachers are not involved in this process in any way: they only receive periodic statistical reports. These reports are not publicly available, since they contain some sensitive information related to both teacher and university. For the analysis of the proposed approach, we extracted the field related to the course evaluation from the four last editions. Before 2016-17 the course was rooted in the Physics M.Sc. degree with a slightly different program, so the student evaluations are not comparable. Each question admits four answers: 1) Unsatisfied, 2) Moderately unsatisfied, 3) Moderately satisfied, 4) Satisfied. The student might also opt for a) Do not want to answer, b) Not relevant. The reports measure students' satisfaction as a percentage, obtained as the ratio between positive answers (3+4) and all the answers (1+2+3+4). The history of students' evaluations is described in Table 2. The parallelization works reported in this manuscript are referred to AY 2018-19; the edition AY 2020-21 is currently ongoing, and no data is available at the time writing. Table 2 seems to suggest that the proposed methodology gathers a wide appreciation from the students, which is the expected effect of tuning the learning challenges to the grain of circles instead of the whole class. A complete evaluation would require the satisfaction to hold also in the following years. We will continue to monitor student evaluation, and we will be ready to dynamically adapt. Dynamic adaptation is one of the cornerstones of the methodology.

Declaration of competing interest

None.

Acknowledgments

The work has been partially supported by the HPC-EUROPA3 project funded under EC H2020 (INFRAIA-2016-1-730897) and the SAPERI project funded by Simularia Srl and Regione Piemonte (POR FESR 2014/2020 - Bando PRISM-e). We gratefully acknowledge the support of Eduardo Quiñones Moreno, from Barcelona Supercomputing Center (BSC), and the computer resources and technical support provided by the BSC; the support of Francesco Iannone from ENEA and the CRESCO/ENEAGRID High Performance Computing infrastructure and its staff [36]; the support of Sergio Rabellino from University of Torino and the Competency Center on Scientific Computing (C3S) and HPC4AI funded by the Region Piedmont POR-FESR 2014-20 (INFRA-P) [5,8]. We thanks Prof. Andrea Mignone, from the University of Torino, for making available the Laplace 2D code produced in his course *Introduction to Parallel Programming with MPI* at the physics department. We thanks Prof. Guy Tremblay (UQAM, CA) for the many suggestions he made on the early versions of this manuscript.

References

[1] M. Aldinucci, M. Danelutto, Stream parallel skeleton optimization, in: Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems, IASTED, ACTA Press, Cambridge, Massachusetts, USA, 1999, pp. 955–962.

[2] M. Aldinucci, M. Danelutto, Algorithmic skeletons meeting grids, Parallel Comput. 32 (7) (2006) 449–462, https://doi.org/10.1016/j.parco.2006.04.001.

[3] M. Aldinucci, S. Campa, M. Danelutto, P. Dazzi, P. Kilpatrick, D. Laforenza, N. Tonellotto, Behavioural skeletons for component autonomic management on grids, in: M. Danelutto, P. Frangopoulou, V. Getov (Eds.), Making Grids Work, CoreGRID, Springer, 2008, pp. 3–16.

[4] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, M. Torquati, Accelerating code on multi-cores with fastflow, in: E. Jeannot, R. Namyst, J. Roman (Eds.), Proc. of 17th Intl. Euro-Par 2011 Parallel Processing, in: LNCS, vol. 6853, Springer, Bordeaux, France, 2011, pp. 170–181.

[5] M. Aldinucci, S. Bagnasco, S. Lusso, P. Pasteris, S. Rabellino, OCCAM: a flexible, multi-purpose and extendable HPC cluster, J. Phys. Conf. Ser. 898 (2017) 082039, https://doi.org/10.1088/1742-6596/898/8/082039.

[6] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati, FastFlow: high-level and efficient streaming on multi-core, in: Parallel and Distributed Computing, John Wiley & Sons, Ltd, 2017, pp. 261–280, Ch. 13, http://dx.doi.org/10.1002/9781119332015.ch13.

[7] M. Aldinucci, M. Danelutto, M. Drocco, P. Kilpatrick, C. Misale, G. Peretti Pezzi, M. Torquati, A parallel pattern for iterative stencil + reduce, J. Supercomput. 74 (11) (2018) 5690–5705, https://doi.org/10.1007/s11227-016-1871-z.

[8] M. Aldinucci, S. Rabellino, M. Pironti, F. Spiga, P. Viviani, M. Drocco, M. Guerzoni, G. Boella, M. Mellia, P. Margara, I. Drago, R. Marturano, G. Marchetto, E. Piccolo, S. Bagnasco, S. Lusso, S. Vallero, G. Attardi, A. Barchiesi, A. Colla, F. Galeazzi, HPC4AI, an AI-on-demand federated platform endeavour, in: ACM Computing Frontiers, Ischia, Italy, 2018, pp. 279–286.

[9] V. Amaral, B. Norberto, M. Goulão, M. Aldinucci, S. Benkner, A. Bracciali, P. Carreira, E. Celms, L. Correia, C. Grelck, H. Karatza, C. Kessler, P. Kilpatrick, H. Martiniano, I. Mavridis, S. Pllana, A. Respício, J. Simão, L. Veiga, A. Visa, Programming languages for data-intensive HPC applications: a systematic mapping study, Parallel Comput. (2019) 102584, https://doi.org/10.1016/j.parco.2019.102584.

[10] G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring), ACM, New York, NY, USA, 1967, pp. 483–485.

[11] A.W. Appel, M. Ginsburg, Modern Compiler Implementation in C, Cambridge University Press, New York, NY, USA, 2004.

[12] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, M. Vanneschi, P3L: a structured high level programming language and its structured support, Concurr. Pract. Experience 7 (3) (1995) 225–255, https://doi.org/10.1002/cpe.4330070305.

[13] U.K. Banerjee, Loop Transformations for Restructuring Compilers: The Foundations, Kluwer Academic Publishers, USA, 1993.

[14] C. Bastoul, Code generation in the polyhedral model is easier than you think, in: 13th International Conference on Parallel Architectures and Compilation Techniques, PACT 2004, 29 September - 3 October 2004, Antibes Juan-les-Pins, France, IEEE Computer Society, 2004, pp. 7–16.

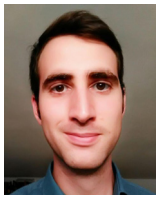
[15] A.J. Bernstein, Analysis of programs for parallel processing, IEEE Trans. Electron. Comput. EC-15 (5) (1966) 757–763, https://doi.org/10.1109/PGEC.1966.264565.

[16] M.A. Bershady, M.A.W. Verheijen, R.A. Swaters, D.R. Andersen, K.B. Westfall, T. Martinsson, The diskmass survey. I. Overview, Astrophys. J. 716 (1) (2010) 198–233, https://doi.org/10.1088/0004-637x/716/1/198.
[17] A. Bisignano, L. Mortarini, E. Ferrero, S. Alessandrini, Model chain for buoyant plume dispersion, Int. J. Environ. Pollut. 62 (2/3/4) (2017), https://doi.org/10.1504/IJEP.2017.089406.
[18] D. Callahan, B.L. Chamberlain, H.P. Zima, The cascade high productivity language, in: Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings, 2004, pp. 52–60.
[19] V. Cesare, I. Colonnelli, M. Aldinucci, Practical parallelization of scientific applications, in: Proc. of 28th Euromicro Intl. Conference on Parallel Distributed and Network-Based Processing, PDP, IEEE, Västerås, Sweden, 2020, pp. 376–384.
[20] V. Cesare, A. Diaferio, T. Matsakos, G. Angus, Dynamics of diskmass survey galaxies in refracted gravity, Astron. Astrophys. 637 (2020) A70, https://doi.org/10.1051/0004-6361/201935950, arXiv:2003.07377.
[21] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar, X10: an object-oriented approach to non-uniform cluster computing, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '05, ACM, New York, NY, USA, 2005, pp. 519–538.
[22] M. Cole, A skeletal approach to exploitation of parallelism, in: Proc. of CONPAR 88, in: British Computer Society Workshop Series, Cambridge University Press, 1989, pp. 667–675.
[23] Intel Corp, Intel threading building blocks, http://software.intel.com/en-us/intel-tbb/, Jul. 2020 (last accessed).
[24] M. Danelutto, M. Torquati, Loop parallelism: a new skeleton perspective on data parallel patterns, in: M. Aldinucci, D. D'Agostino, P. Kilpatrick (Eds.), Proc. of Intl. Euromicro PDP 2014: Parallel Distributed and Network-Based Processing, IEEE, Torino, Italy, 2014, pp. 376–384.
[25] M. Danelutto, J.D. García, L.M. Sánchez, R. Sotomayor, M. Torquati, Introducing parallelism by using REPARA C++11 attributes, in: 24th Euromicro Intl. Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion,Crete, Greece, February 17-19, 2016, IEEE Computer Society, 2016, pp. 354–358.
[26] A. Darte, Y. Robert, F. Vivien, Loop parallelization algorithms, in: S. Pande, D.P. Agrawal (Eds.), Compiler Optimizations for Scalable Parallel Systems Languages, Compilation Techniques, and Run Time Systems, in: Lecture Notes in Computer Science, vol. 1808, Springer, 2001, pp. 141–172.
[27] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, in: Usenix OSDI '04, 2004, pp. 137–150.
[28] D. del Rio Astorga, M.F. Dolz, L.M. Sánchez, J.D. García, M. Danelutto, M. Torquati, Finding parallel patterns through static analysis in C++ applications, Int. J. High Perform. Comput. Appl. 32 (6) (2018), https://doi.org/10.1177/1094342017695639.
[29] M. Drocco, V.G. Castellana, M. Minutoli, Practical distributed programming in C++, in: M. Parashar, V. Vlassov, D.E. Irwin, K. Mohror (Eds.), HPDC '20: the 29th International Symposium on High-Performance Parallel and Distributed Computing, Stockholm, Sweden, June 23-26, 2020, ACM, 2020, pp. 35–39.
[30] J. Enmyren, C.W. Kessler, SkePU: a multi-backend skeleton programming library for multi-GPU systems, in: Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications, HLPP '10, ACM, New York, NY, USA, 2010, pp. 5–14.
[31] S.M. Ghazimirsaeed, S.H. Mirsadeghi, A. Afsahi, An efficient collaborative communication mechanism for MPI neighborhood collectives, in: 2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2019, pp. 781–792.
[32] H. González-Vélez, M. Leyton, A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers, Softw. Pract. Exp. 40 (12) (2010) 1135–1160, https://doi.org/10.1002/spe.1026.
[33] J.L. Gustafson, Reevaluating Amdahl's law, Commun. ACM 31 (5) (1988) 532–533, https://doi.org/10.1145/42411.42415.
[34] M. Harris, CUDA pro tip: write flexible kernels with grid-stride loops https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/, 2021 (last accessed).
[35] N.R. Herbst, S. Kounev, R. Reussner, Elasticity in cloud computing: what it is, and what it is not, in: Proc. of the 10th Intl. Conference on Autonomic Computing, USENIX, San Jose, CA, 2013, pp. 23–27.
[36] F. Iannone, F. Ambrosino, G. Bracco, M. De Rosa, A. Funel, G. Guarnieri, S. Migliori, F. Palombi, G. Ponti, G. Santomauro, P. Procacci, CRESCO ENEA HPC clusters: a working example of a multifabric GPFS spectrum scale layout, in: 2019 International Conference on High Performance Computing Simulation, HPCS, 2019, pp. 1051–1052.
[37] Intel, Intel® AVX-512 instructions, https://software.intel.com/en-us/blogs/2013/avx-512-instructions, 2013.
[38] F. Irigoin, R. Triolet, Supernode partitioning, in: Proc. of the 15th ACM Symposium on Principles of Programming Languages, POPL, ACM, New York, NY, USA, 1988, pp. 319–329.
[39] K. Kennedy, C. Koelbel, H. Zima, The rise and fall of high performance Fortran: an historical object lesson, in: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, HOPL III, Association for Computing Machinery, New York, NY, USA, 2007, pp. 7–1–7–22.
[40] I. Koren, C.M. Krishna, Fault-Tolerant Systems, 1st edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
[41] C. Lengauer, Loop parallelization in the polytope model, in: Proc. of the 4th Intl. Conference on Concurrency Theory, CONCUR, in: LNCS, vol. 715, Springer, Hildesheim, Germany, 1993, pp. 398–416.
[42] A. Mignone, G. Bodo, S. Massaglia, T. Matsakos, O. Tesileanu, C. Zanni, A. Ferrari, PLUTO: a numerical code for computational astrophysics, Astrophys. J. Suppl. Ser. 170 (1) (2007) 228–242, https://doi.org/10.1086/513316.
[43] C. Misale, M. Drocco, M. Aldinucci, G. Tremblay, A comparison of big data frameworks on a layered dataflow model, Parallel Process. Lett. 27 (01) (2017) 1–20, https://doi.org/10.1142/S0129626417400035.
[44] Mpi forum, https://www.mpi-forum.org, 2020 (last accessed).
[45] J. Neal, T. Fewtrell, M. Trigg, Parallelisation of storage cell flood models using OpenMP, Environ. Model. Softw. 24 (7) (2009) 872–877, https://doi.org/10.1016/j.envsoft.2008.12.004.
[46] M.F.P. O'Boyle, P.M.W. Knijnenburg, Integrating loop and data transformations for global optimization, in: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, Paris, France, October 12-18, 1998, IEEE Computer Society, 1998, p. 12.
[47] Khronos Compute Working Group, OpenACC directives for accelerators, http://www.openacc-standard.org, Nov. 2012.
[48] I. Park, M.J. Voss, S.W. Kim, R. Eigenmann, Parallel programming environment for OpenMP, Sci. Program. 9 (2001) 143–161.
[49] Y. Tanaka, K. Taura, M. Sato, A. Yonezawa, Performance evaluation of OpenMP applications with nested parallelism, in: Languages, Compilers, and Run-Time Systems for Scalable Computers, Springer, 2000, pp. 100–112.
[50] M. Tchiboukdjian, N. Gast, D. Trystram, J.-L. Roch, J. Bernard, A tighter analysis of work stealing, in: O. Cheong, K.-Y. Chwa, K. Park (Eds.), Algorithms and Computation, Springer, Berlin, Heidelberg, 2010, pp. 291–302.
[51] C. Terboven, A. Spiegel, D. an Mey, S. Gross, V. Reichelt, Experiences with the OpenMP parallelization of DROPS, a Navier-Stokes solver written in C++, in: OpenMP Shared Memory Parallel Programming - International Workshops, IWOMP 2005 and IWOMP 2006, Eugene, OR, USA, June 1-4, 2005, Reims, France, June 12-15, 2006. Proceedings, in: LNCS, vol. 4315, Springer, 2005, pp. 95–106.
[52] E. Tomasi, L. Giovannini, M. Falocchi, G. Antonacci, P.A. Jiménez, B. Kosovic, S. Alessandrini, D. Zardi, L. Delle Monache, E. Ferrero, Turbulence parameterizations for dispersion in sub-kilometer horizontally non-homogeneous flows, Atmos. Res. 228 (2019) 122–136, https://doi.org/10.1016/j.atmosres.2019.05.018.
[53] J.L. Träff, S. Hunold, Cartesian collective communication, in: Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Association for Computing Machinery, New York, NY, USA, 2019, pp. 1–11.
[54] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103?111, https://doi.org/10.1145/79173.79181.
[55] P. Viviani, M. Drocco, D. Baccega, I. Colonnelli, M. Aldinucci, Deep learning at scale, in: Proc. of 27th Euromicro Intl. Conference on Parallel Distributed and Network-Based Processing, PDP, IEEE, Pavia, Italy, 2019, pp. 124–131.
[56] J. Weidendorfer, M. Kowarschik, C. Trinitis, A tool suite for simulation based analysis of memory access behavior, in: Proc. of 4th Intl. Conference on Computational Science, ICCS, in: LNCS, vol. 3038, Springer, Kraków, Poland, 2004, pp. 440–447.
[57] B. Wilkinson, M. Allen, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, 2nd edition, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.
[58] N. Wirth, Algorithms and Data Structures, Prentice Hall, 1985.
[59] H. Zhong, M. Mehrara, S. Lieberman, S. Mahlke, Uncovering hidden loop level parallelism in sequential applications, in: 2008 IEEE 14th International Symposium on High Performance Computer Architecture, 2008, pp. 290–301.

Marco Aldinucci is a full professor and the P.I. of the Parallel Computing research group at the University of Torino. He received his Ph.D. from the University of Pisa (2003), and he has been a researcher at the Italian National Research Council (CNR). He is the author of over 120+ scientific articles. He is the recipient of the HPC Advisory Council University Award 2011, the NVidia Research award 2013, the IBM Faculty Award 2015, the Autodesk award 2021. He has participated in over 15 EU-funded research projects on parallel and cloud computing attracting over 6M€ of research funds to the University of Torino. In 2018, he incepted the HPC4AI Turin's competency center on HPC-AI convergence. From 2021, he is the founding director of the CINI "HPC Key Technologies and Tools" national laboratory, gathering researchers from 35 Italian Universities. He is a member of the Governing Board of the EuroHPC Joint Undertaking. He is a co-designer of the FastFlow programming framework and several other programming frameworks and libraries for parallel computing.

Valentina Cesare is a Ph.D student in Physics at the Physics Department of the University of Turin. Her Ph.D supervisor is Prof. Antonaldo Diaferio and her Ph.D thesis is about the investigation of the dynamics of disk and elliptical galaxies with the theory of modified gravity Refracted Gravity.

Iacopo Colonnelli is a Ph.D. student in Modeling and Data Science at Universita di Torino. He received his master's degree in Computer Engineering from Politecnico di Torino with a thesis on a high-performance parallel tracking algorithm for the ALICE experiment at CERN. His research focuses on both statistical and computational aspects of data analysis at large scale and on workflow modeling and management in heterogeneous distributed architectures.

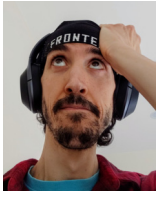


Alberto Riccardo Martinelli received his Master's Degree in 2020 with a master thesis on CAPIO (Cross-Applicaton Programmable I/O), a novel method for in-transit HPC computing. While he was a MSc student he worked as a junior research engineer in the parallel computing research group of the university. He is currently PhD student at the Computer Science Department of the University of Turin working on storage for HPC, parallel computing, distributed computing and heterogeneous computing.

Gianluca Mittone received his Bachelor's Degree in 2017 with a thesis on the handling of exceptions in Description Logics, proposing the implementation of an algorithm for the automatic revision of ontologies exploiting a Typicality operator, and his Master's Degree in 2019 with a master thesis on a novel distributed approach for deep learning, named NNT (Nearest Neighbors Training), which take advantage of a locally synchronous approach in order to achieve a better trade-off between computational time and learning results. He is currently PhD student at the Computer Science Department of the University of Turin working on different projects involving HPC and Machine Learning techniques.

Barbara Cantalupo is a Research Engineer at the Computer Science Department of the University of Torino. She received her master's degree in Computer Science from the University of Pisa (1994), and she has been a researcher in the parallel computing field at University and at the Italian National Research Council (CNR). Afterwards, she has worked for 15 years in several private companies in the area of supercomputing, mobile networks and space, acquiring knowledge on different application fields.

Carlo Cavazzoni is a leading expert in high performance computing and related policies in Europe. Graduated in Physics at Uni Modena, PhD at SISSA in Trieste. For many years he has coordinated the activities regarding the evolution of supercomputer architectures, programming models and related infrastructures at CINECA, the Italian supercomputing Centre. He is currently Head of Computational R&D, and Director of the HPC Lab at Leonardo SpA, and sits in European boards within EuroHPC and ETP4HPC.



Maurizio Drocco is a Postdoctoral Research Associate at the Pacific Northwest National Laboratory. He received his Ph.D. in Computer Science from the University of Torino in October 2017, with a thesis about distributed programming in C++.

He has been Research Intern at IBM Thomas J. Watson Lab (NY) and at IBM Dublin Research Lab, and Research Associate at University of Torino since 2009. He has co-authored papers in international journals and peer- reviewed conference proceedings (Google h-index 8). His research concerns cross-stack parallel and distributed computing: lock-free synchronization, memory allocators, memory models, up to high-level programming abstractions and semantics.