Practical Parallelization of Scientific Applications

Valentina Cesare, Iacopo Colonnelli, Marco Aldinucci
Physics and Computer Science Departments
University of Turin
Turin, Italy
e-mail: {valentina.cesare,iacopo.colonnelli,marco.aldinucci}@unito.it

***Abstract*—This work aims at distilling a systematic methodology to modernize existing sequential scientific codes with a limited re-designing effort, turning an old codebase into *modern* code, i.e., parallel and robust code. We propose an automatable methodology to parallelize scientific applications designed with a purely sequential programming mindset, thus possibly using global variables, aliasing, random number generators, and stateful functions. We demonstrate the methodology by way of an astrophysical application, where we model at the same time the kinematic profiles of 30 disk galaxies with a Monte Carlo Markov Chain (MCMC), which is sequential by definition. The parallel code exhibits a 12 times speedup on a 48-core platform.**

***Keywords*-loop parallelism; checkpointing; scientific code; openMP**

I. Introduction

The shift toward multi-core and many-core technologies 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. To date, however, parallel programming has not embraced much more than low-level libraries, which often require the architectural re-design of the application. While re-designing with an explicitly parallel approach is still the most effective option to achieve scalable and efficient parallel codes, this approach is unable to effectively support the mainstream of software development that builds on legacy sequential codebases. In this context, human productivity and time-to-solution are equally, if not more, important aspects than performance. Also, re-design is a treacherous activity because it might impair the correctness of the code; typical pitfalls are in numerical stability, validation of results, and usage of random number generators in parallel codes.

In the last three decades, parallel programming methodologies significantly evolved, aiming at easing programmers' tasks and improving program efficiency. The common thread of this evolution has been the raising of the level of abstraction of concurrency management primitives. A crucial step in this process has been the definition of algorithmic paradigms or *skeletons* (as called by M. Cole in the eighties [1]–[6]) 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 [7], the *data parallel* paradigm (map and reduce skeletons) enabling the *MapReduce* programming model [8] with its whole array of Apache-BigData solutions and the *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 compliance of the embedded sequential code with programming model constraints, such as associativity of accumulation operations, concurrent access to shared data structures, absence of a persistent state in pure functions. Parallel frameworks, offering programming patterns, allow for the *explicit* expression of concurrency in applications to better exploit parallel hardware. Notwithstanding, a large portion of production software, from a broad range of scientific and industrial areas, is still developed sequentially.

This kind of abstraction significantly simplifies the hand-coding of applications, but it is still too low-level to effectively automatize the optimization of the parallel code: here the major weakness lies in the lack of information concerning the intent of the code (idiom recognition [9]); inter-procedural/component optimization further exacerbates the problem. The generative approach focuses on synthesizing implementations from higher-level specifications rather than transforming them. The programmer can directly define these specifications, which, in the form of either higher-order functions (as in Intel TBB) or directives (as in OpenMP), can be associated with semantically meaningful points of the code, such as function calls and loops. Programmer specifications enable the generation of reasonable parallel codes, at least in the shared-memory model. 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 to enforce compilation constraints or specific code generation, including parallelization [10].

Among the mentioned meaningful points of the code, loops are certainly the most common in scientific codes. They are typically used to navigate arrays and to go through discretization of dimensions (e.g., time and random walks).

Since several tools to parallelize a single loop exist, a cost-effective method enhancing the performance and the robustness of an entire scientific application revolves around three main tasks. Firstly, selecting which loops we can correctly parallelize while avoiding both to restrict parallelism needlessly and to require complex code transformations that might affect the numerical stability of the code. Secondly, selecting which loops are worth to be parallelized. Thirdly, selecting checkpoints at the beginning of a nonparallelizable loop that does not appear within a parallel loop. In these points, the execution is sequential, and all the data structures are globally consistent and can be quickly resumed.

This work directly aims at codifying a practical guide for the cost-effective parallelization of scientific applications in the shared-memory model. The approach is intentionally practical and intended to guide both domain expert practitioners and students wanting to parallelize their codes without necessarily becoming parallel computing experts. After a brief outline of related work (Sec. II), we will discuss the methodology in Sec. III, followed by a sample application to an astrophysics problem (Sec. IV) and the corresponding performance evaluations (Sec. V). Finally, Sec. VI concludes the article.

II. Related work

One of the main issues when dealing with loop parallelism is to find a solid strategy for the *iterations scheduling* over a set of parallel workers, which must ensure excellent performances in a wide range of cases while ensuring sequential equivalence (at least to the extent allowed by the finite numerical precision of floating-point operations). Since the nineties, the *polytope model* has been recognized as the de-facto way to map data dependency graphs to processing elements topologies in the context of parallel loops [11]. Indeed, the rich mathematical framework introduced by lattice theory allows investigating the space of possible scheduling strategies.

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 interface comes in the form of either higher-order functions, as in Intel TBB, or `#pragma` directives, as in OpenMP or OpenACC [12]. This kind of approach has the advantage of minimizing the learning curve for developers with little or no experience in parallel programming while allowing more expert users to fine-tune their applications by modifying many optional parameters. For this, these techniques have been preferred to lower-level alternatives over time, as explicitly parallel approaches, which provide more scalability at the cost of a less intuitive interface [13].

The massive amount of sequential codebases pushed the parallel computing community to find suitable techniques to provide a fully *automatic parallelization* of serial codes [14], [15]. The main drawback of this kind of approach is that, to ensure correctness in all cases, often some straightforwardly removable dependencies prevent full exploitation of the potential parallelism degree. As a result, the performance improvement against the baseline is often modest. Unfortunately, even hybrid approaches like OpenMP can often perform much below the expectations, due to the inability of the underlying runtime in finding the best way to rearrange data dependencies. Conversely, a minor and straightforward reorganization of some small portions of the code can significantly improve the overall performances. Nevertheless, a significant gap exists between the toy examples commonly provided in OpenMP tutorials and guidelines and real scientific applications with multi-level nested loops and, potentially, parallel random number generation. The aim of this work is precisely to fill this gap, providing a practical but generic enough methodology for loop parallelization of a typical scientific code. Even if in literature there are some examples of the application of OpenMP to serial scientific codes [16], [17], the discussion is usually very focused on the analysed 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, trying to find several different parallel patterns in the code, rather than simply loops [18]. Tools as *Parallel Pattern Analyzer Tool* [19] aim at identifying 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).

III. Methods

The most prominent control flow statement in scientific codes is the *loop*, which denotes iterative computations. Pragmatically, the *for-loop* iterates over arrays of data, whereas the *while loop* iterates up to a given convergence criterion [20]. They can appear juxtaposed in a sequence or nested in any order. Code in the loop body might exhibit a network of dependencies among different loops and iterations of the same loop. Examples are in:

- *Particle simulations*, in which an internal loop computes quantities related to each particle and an external loop advances the simulation time step.
- *Optimization algorithms*, in which 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.
- *ODE solvers*, in which internal loops iterate over different functions or subsystems and an external loop advances the time step.

To parallelize a nested loop program, we advocate the following methodology:

1) Identify all parallelizable loops in the code, according to a depth-first search strategy.

2) Evaluate the potential performance gain obtainable 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 nonparallelizable loops as a reference to implement a checkpointing logic, to support stop-resume behaviour.

We will detail these steps in the remaining sections.

A. 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 additional relations to turn the graph into a tree. A good example is the domination relationship [21], which induces the *loop-nest tree*, in which each node refers to a distinct loop and a node B is a child of node A if they are nested loops, 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 analysing the code. Such representation of multi-level nested loops suggests a parallelizing strategy consisting in considering one loop at a time with a *depth-first search* technique, 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, ensuring to preserve potential inter-loop dependencies.

Bernstein's seminal paper clearly states that the problem of determining if two arbitrary program sections are parallelizable is undecidable and 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* [22].

We can categorized loops as:

- *Loop-independent*, when iteration i does not depend on any iteration $j < i, \forall i < N$.
- *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 in the presence of loop-carried dependencies things get more involved. Unfortunately, loops that are not written with a parallel mindset can sometimes contain unnecessary dependencies. They 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 [11]. The parallelization of loops that have loop-carried dependencies in their original form is often possible by transforming the loop into some new form in which the dependencies are removed or arranged to occur at a sufficient distance so that concurrent iterations do not conflict. 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 for updating a variable with the next random number in a sequence.

A practical way to parallelize a program is to descend the loop hierarchy until a parallelizable loop is found. Unfortunately, things can get a bit more complicated in the presence of conditional branches, which can make the parallelizability of a loop a data-dependent property. In this case, it is useful to identify the most computationally demanding paths in the control flow graph and separate them from the rest of the code in dedicated procedure calls to reduce complexity. 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 nonparallelizable cases, together with one or more parallel versions for the others.

B. 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, which always comes with a certain amount of overhead introduced by synchronizations among different workers. When the effective computation time of a parallel section (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 which can be considered on the order of tens of thousands of clock cycles. Finer grains can be addressed only with lock-free programming frameworks, such as Fastflow, that can support grains down to hundreds of clock cycles [5].

Some useful quantities that should be taken into account when estimating the potential performance gain introduced by the parallelization of a specific loop are the *number of iterations*, which affect the maximum obtainable degree of parallelism, and the *total time* spent by the program inside the loop, which determines the maximum achievable speedup. In some cases, 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 [23] can be considerably helpful.

In general, it would be better to parallelize an outer loop instead of one of its nested counterparts, because this strategy minimizes the introduced overheads, e.g., for thread creation and synchronization, even if we cannot define a better strategy. For example, if a loop has very few iterations, the parallelization of one or more of its inner loops can lead to better results and, if the code runs on many processors, the parallelization of both inner and outer loops can be more convenient [24]. Nevertheless, an effective greedy technique would be to start parallelizing the suitable outermost loop, where suitable means both feasible and convenient, and then descend the loop hierarchy 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 to evaluate the maximum performance gain that can be expected after the effective transformation of the program. To predict the performance of a parallel code, we have to investigate its potential strong and weak scalability. *Strong scaling* represents the ability of a software to solve a problem of fixed size faster with a higher amount of computing resources and it is strictly related to the notion of *speedup* of a program. The speedup S is defined as the ratio between the time t_s taken by the sequential code and the time $t_\mathrm{p}(n)$ taken by the parallel code with an increasing 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 [25], an upper bound for a program speedup can be expressed as the inverse of $s + p/n$, where s is the fraction of the total execution time of the code spent by the serial portion of the code, and p is the fraction of the total execution time of the code spent by the parallelized part. Unfortunately, Amdahl's law does not take into account 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 a problem of variable size, keeping constant the amount of work assigned to each computing resource. Gustafson's law, formulated in 1988 [26], describes the *scaled speedup* as $s + p \times n$. This law states that the size of a problem scales with the available number of processors: the time spent in the parallel part of the code linearly grows with processors, whereas the time spent in the serial part of the code remains constant with the size of the problem. This means that, if a code is fully parallelizable, the time spent by a problem of size n to run on n processors will remain constant. The scaled speedup does not have an upper limit.

C. Make loops self-contained

Once we have identified a loop that is worth to parallelize, it is necessary to transform the iterative construct into a self-contained procedure call, which takes in input all the externally declared variables used inside an iteration. If the code is parallelized on a many-core accelerator with a local address space, all the external variables have to be passed by value as arguments of the newly created procedure. Otherwise, this is necessary only for the variables that have to be written inside the body of the procedure. Nevertheless, some modern programming models for hardware accelerators (such as latest versions of CUDA) provide an abstract unified address space between host and device memory and manage data transfers under the hood, considerably reducing the programming effort.

Frequently a loop is used to iterate over an array of inputs to produce an array of outputs, but it is not uncommon that such loop is immediately followed by another loop that combines all the produced elements in a single value, by means of 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. With an input array of n elements and n workers, a reduce pattern is able to produce the final output in $\log n$ 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 second approach since it is much easier and faster to implement and can guarantee better performance portability among different hardware architectures. Instead, we can resort to a manual implementation only when appropriate.

D. Deal with random number generators

The complexity in dealing with random number generators in parallel codes is due to their stateful nature, which is primarily aimed at approximating genuinely random numbers with actually deterministic numbers generated with pseudo-random number generators (PRNG). These numbers can be reproduced if the state of the PRNG is known. PRNGs appearing in sections of code that are sequentially executed do not need special care but rather the moment of generation can be used as a checkpoint (see Sec. III-E). On the contrary, their parallelization requires special care. Firstly, the PRNG implementation should be *thread-safe*, i.e., performed from multiple threads safely and *reentrant*, i.e., performed from multiple actors of the same thread safely (called multiple times within the same thread). Secondly, to enforce reproducibility, the random sequence generated in each parallel section should be deterministic, thus independent of the relative execution order of the parallel sections. This is achieved by privatizing the random induction variable. In object-oriented languages, this can be easily achieved by using an array of PRNG objects. Thirdly, to enforce correctness and reproducibility, the array of PRNG objects should be initialized with a seed generated with a master PRNG implemented with another algorithm, since using the same algorithm is going to reduce the period and induce loss of uniformity of distribution in generated numbers. Once the seed of the master PRNG is fixed, the sequence of random numbers generated in each parallel section should be deterministic. Fourthly, scientists 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 scientist's duty to prove that the sequential and the parallel codes compute the same stochastic process.

E. Implement checkpointing logic

The sequential regions of a parallel code do not provide a gain in performance, but they can provide another advantage. Indeed, since they define a global order in the execution of the program, they can be defined as checkpoints. A *checkpoint* is a snapshot of the entire state of the process at the moment it was taken, which represents all the information needed to restart the process from that point [27]. Usually, checkpoints are recorded on *stable storage* that is persistent storage with some reliability requirements.

Two essential concepts related to checkpoints are the *checkpointing overhead*, i.e., the increase in the total execution time caused by the introduction of the checkpointing procedure, and the *checkpointing latency*, i.e., the time needed to save the checkpoint. The aim of an appropriate checkpointing strategy is to minimize the first quantity. In order to do that, two different approaches are possible. The first is to try 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.

As an example, a checkpoint can be defined in a random sequence of numbers initialized with a given seed. If the application fail-stops in 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 permanent storage. In this work, we can see that this procedure is particularly useful in Monte Carlo Markov Chains (MCMCs), that can be quite computationally expensive.

IV. Application to astrophysics

We apply the parallelization procedure described above to an astrophysical case. We model, at the same time, the rotation curves and the vertical velocity dispersions from the mass distributions of 30 disk galaxies belonging to the Disk Mass Survey [28], exploring the agreement between the models and the measured data with a Bayesian approach. We run a MCMC for 19000 iterations after 1000 burn-in steps, a number suitable to reach a good convergence.

We adopt flat priors for the free parameters of the model and a Metropolis-Hastings acceptance criterion for the MCMC, obtaining the random variate $\vec{x}$ at the step $i+1$ from the multi-variate Gaussian probability density $G(\vec{x}|\vec{x}_i)$ peaked at $\vec{x}_i$, the random variate at the previous MCMC step. We define a Metropolis-Hasting ratio

$$R_{\rm MH} = \frac{p(\vec{x}) \times \mathcal{L}(\vec{x})}{p(\vec{x}_i) \times \mathcal{L}(\vec{x}_i)} \frac{G(\vec{x}|\vec{x}_i)}{G(\vec{x}_i|\vec{x})} \ , \qquad (1)$$

where $\vec{x}$ is the free parameters vector, $p(\vec{x})$ is the product of the priors of the parameters and $\mathcal{L}(\vec{x}) = \sqrt{\exp\left[-\chi^2_{\rm tot}(\vec{x})\right]}$ is the likelihood. If $R_{\rm MH} \geq 1$ $\vec{x}$ is accepted, else it is accepted with probability $R_{\rm MH}$ or rejected with probability $1 - R_{\rm MH}$.

A. Serial version

The overall structure of the code, written in C++, is reported in Fig. 1. A first preparatory step imports some input data from external files. Among them we find, for each galaxy, the mass density, the kinematic data, that have to be compared with the models, and the features of the grid where the gravitational potential, the gravitational field and the kinematic profiles of the galaxy are computed. Right after the data import, we define the priors and initialize the MCMC.

The main body of the program consists of two nested for-loops. The most external loop iterates on the number of MCMC iterations we decide to perform, while the internal one iterates on the number of galaxies present in the sample. Within the second level for-loop we have to execute the same sequence of operations for every galaxy:

1) Computation of the galaxy's gravitational potential ϕ from its mass density ρ, solving the Poisson equation

$$\nabla^2 \phi(R,z) + S(\rho; R, z) = 0 \qquad (2)$$

with a Successive Over Relaxation Poisson solver [29], where $S(\rho; R, z)$ is the *source term.*

2) Computation of the radial and vertical derivatives of ϕ (gravitational field).
3) Computation of the rotation curve and the vertical velocity dispersion profile and their respective χ^2.

After that, the χ^2 of the rotation curve and of the vertical velocity dispersion computed for each galaxy are reduced

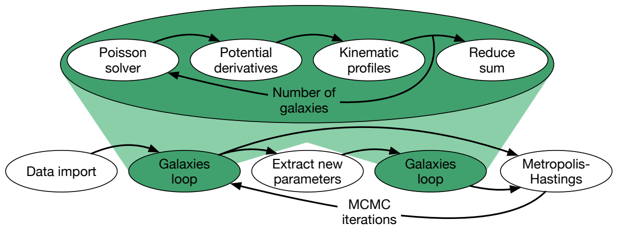

Figure 1. Flowchart for the serial version of the program.

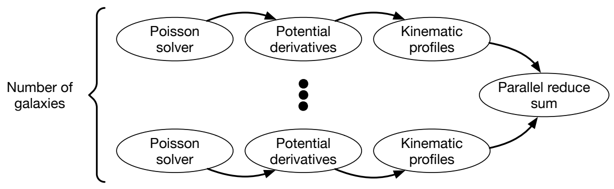

Figure 2. Flowchart for the parallel version of the galaxies loop.

(with sum operation) to obtain the global likelihood. Then, both the inner for-loop and the reduce operation are repeated for the second part of the MCMC, starting from a new combination of free parameters randomly generated from the previous one. Finally, a Metropolis-Hastings acceptance criterion is applied to accept or reject the last combination of parameters for the next iteration of the MCMC.

B. Parallel implementation

Since this code involves many galaxies, whose quantities are discretized, for each of them, on a grid of 100-150k points, its sequential execution can become very slow when several MCMC iterations are required to reach a good convergence. In particular, the most computationally expensive region of the code is the Poisson solver, which we have to run 2×30 times per MCMC iteration.

For this reason, it is worth to parallelize this code following the methods explained in Sec. III. First of all, we identify the code regions with true data dependencies. The first important region is the MCMC, which is a sequential process by definition as the new combination of free parameters is drawn from the previous one at every step. This region cannot, therefore, be parallelized, but it can be used for checkpointing purposes (more details are in Sec. IV-C). The second region is made of the innermost loops, which compute the field of every galaxy from the potential and the kinematic profiles from the field.

Then, we identify the points without true data dependencies, which can be parallelized. This region is the second level for-loop that iterates on the number of galaxies in the sample. Since we independently perform the same operations for every galaxy, this loop results *embarrassingly parallelizable*. Also, the reduce operation can be parallelized, implying an eliminable dependence since it is based on an associative binary operator (the sum in this case) on the χ^2 derived from the kinematic profiles of each galaxy. For this reason, we put the reduce operation within the for-loop that iterates on the number of galaxies. We parallelize this for-loop using the OpenMP library, with the `#pragma omp parallel for shared(`χ^2`) reduction(+:` $\chi^2_{\rm tot}$`)` directive, obtaining the structure reported in Fig. 2.

C. Checkpoints

The MCMC can be very computationally demanding, requiring an execution time of one week on a modern platform even after an adequate optimization process. During such a long period, the code can be interrupted for many reasons and restarting it from the beginning after every stop would result in a severe loss of time.

In our MCMC, we implement two random sequences initialized with the same seed. The first is the sequence for a random number U, that follows a real uniform distribution between 0 and 1. Its definition is necessary for the Metropolis-Hastings criterion: if $R_{\rm MH} \geq U$ $\vec{x}_{i+1} = \vec{x}$, else $\vec{x}_{i+1} = \vec{x}_i$. The second is the sequence for a multi-variate random variable that follows a normal distribution peaked on the previous combination of free parameters with a given standard deviation, used to generate the new combination of free parameters from the previous one.

To implement these two random sequences, we define two pseudo uniform random number generators, G_1 and G_2, from the C++ library Boost. Every 1000 iterations we print in text files, using the `ofstream` operator, the following information: 1) The values of the generators G_1 and G_2 at the beginning of the MCMC iteration; 2) The parameters chains at the end of the MCMC iteration. If the code is interrupted between the MCMC iterations 7000 and 8000, before the MCMC for-loop we import, using the `ifstream` operator, the chains made of the first 7000 parameters and the two generators saved at iteration 7000. At this point, it is sufficient to start the MCMC for-loop at iteration 7000, so we can complete the chain without restarting it from the beginning.

V. Performance Evaluation

We investigate the strong and the weak scaling of this code to test its performance. To perform these tests, we only take 5 MCMC iterations to operate in reasonable timescales. We perform the tests on a Linux OS running on a 48-core platform (quad-socket Intel Xeon E7, 12 physical cores per socket) with 768 GB of memory. The following performance analysis also includes the impact of the *numa control* on the results. On UNIX systems, a process can be launched with the `numactl` command, which spreads the computation on different sockets and affects how data are stored in different cache levels. In particular, the *interleave all* mode 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* mode uses one socket per time, until the number of threads saturates its cores. For comparison, we also investigate the strong and the weak scaling of the code on a Linux OS running on a 24-core platform (two-socket Intel Xeon E5, 12 physical cores per socket) with 128 GB of memory, using the default policy of the machine (without the *numa control*).

A. *Strong scaling*

To investigate the strong scaling of this code, we consider the entire galaxy sample. To measure the actual speedup of our code we take the CPU time of each MCMC iteration with the function `gettimeofday` (μs) setting in the latter an increasing number of threads from 1 to 48 with the `omp_set_num_threads(Nthreads)` function, where the number of threads `Nthreads` is taken in input from an external file.

The left panel of Fig. 3 shows the strong scaling (speedup) with the *numa control* in the *block* mode (red line) and in the *interleave all* mode (blue line) and with the default policy of the machine (green line) on the 48-core platform, where $t_{\rm s}$ and $t_{\rm p}$ are the execution times for the sequential and the parallel code, respectively. Each point of the plot is the average over 5 MCMC runs. The errors on the ratio $t_{\rm s}/t_{\rm p}$, shown in the figure as shaded areas, are calculated by propagating the uncertainties on $t_{\rm s}$ and $t_{\rm p}$, which in turn are the standard deviations of the times of the five respective iterations.

This figure shows that the ideal law $S = N_{\rm threads}$ holds more or less from 1 to 4 threads, but there is still a quite good linear trend until 9-10 threads. Then, the measures start to increase more slowly, converging to an asymptotic value around 12. According to the Amdahl's law, this means that the time of the serial fraction of the code is about $1/12$ of the total computational time. However, this asymptotic value is reached in different ways in the three modes.

The speedup of the code is comparable among the three modes only for a number of threads between 1 and 9 and between 35 and 48. For an intermediate number of threads, the speedup in the *block* mode increases more slowly than in the other two cases. This means that, if we operate in the *block* mode, we need a number of threads equal to the resources of the entire machine, or of a large part of it, to obtain the maximum gain in performance. Instead, with the default policy of the machine and in the *interleave all* mode, whose strong scaling curves are very similar, we only need a number of threads equal to half of the resources of the machine. Between 10 and 34 threads, the default and the *interleave all* modes provide a better gain in performance than the *block* mode.

From the left panel of Fig. 3 we can notice the big jump in performance between 24 and 25 threads in the default and in the *interleave all* modes. In the *block* mode, we can observe an abrupt slowdown of the parallel code in correspondence of 13 threads, which is where the number of threads has saturated the first socket and has just started to use the second one.

The left panel of Fig. 4 shows the strong scaling with the default policy of the machine on the 24-core platform. Qualitatively the trend of this strong scaling curve is quite similar to the corresponding one on the 48-core platform. Nevertheless, the most striking difference is the asymptotic value reached by the speedup that on the 24-core platform is around 8 and on the 48-core platform is around 12.

B. *Weak scaling*

In order to investigate weak scaling, we have to define a unit of work. Since the parallelized for-loop iterates on the number of galaxies in the considered sample, so that the larger the number of galaxies, the more computational demanding the code, we define *one galaxy* as our unit of work. At this point, we measure the time spent by the program with n galaxies run on n threads, where n goes from 1 to 48. The galaxies in our original sample are all different between each other. Nevertheless, to consider homogeneous units of work, we choose a single galaxy and we run the related computation once on a single-core, twice on two cores and so forth.

In the right panel of Fig. 3 we plot the mean time in seconds of each MCMC iteration in function of the number of used threads, which coincides with the number of used galaxies, for the default (green line), the *interleave all* (blue line) and the *block* modes (red line). As in the left panel of the same figure, the shaded areas show the error bars of the measures, taken as the standard deviations of the times of the five respective iterations. If the code were fully parallelizable the time would remain constant, for whichever number of threads equal to the number of galaxies but there is often a section of code which remains sequential which makes the trend not perfectly constant.

We can observe different trends in the three modes. As for the strong scaling, the weak scaling curves corresponding to the default and the *interleave all* modes are quite similar. With these two modes, there is a very gradual slowdown of the code from 1 to 37 threads, where the time passes from 4.9 to 5.9 seconds in the *interleave all* mode and from 4.9 to 6.2 seconds in the default mode. We can thus state that weak scalability is mostly satisfied in this range of threads. Then the code shows a steeper slowdown from 37 to 48 threads, where we lose almost two seconds in performance, passing from ~ 6 to 7.7 or 7.9 seconds, in the *interleave all* and in the default modes, respectively. We can see regular bumps of the time from 4 to 37 threads.

In the *block* mode, we observe almost the same global trend as in the default and in the *interleave all* modes, but passing from 1 to 14 threads, which means that the slowdown of the code occurs faster. After 14 threads, the mean time of each MCMC iteration remains almost constant, around 7.5 seconds, and the standard deviation of the measures increases. Also this test shows that the default and the *interleave all* modes provide the highest efficiency, which means that to obtain the best performance gain we have to use the entire machine. In this test, the *interleave all* mode appears slightly more efficient than the default mode. Conversely, using only a part of the machine sufficient to host the number of used threads, as in the *block* mode, leads to a loss in performance.

The right panel of Fig. 4 shows the weak scaling in the default policy of the machine on the 24-core platform. The time remains mostly constant from 1 to 10 threads, and it shows a gradual slowdown until 24 threads, passing along the entire considered range of threads from 4.7 to 5.8 seconds. So, the default mode on the 24-core platform from 1 to 24 threads almost behaves like the corresponding curve on the 48-core platform between 1 and 37 threads. The default curve on the 48-core platform appears flatter than the corresponding curve on the 24-core platform between 1 and 24 threads.

We can state that our code satisfies weak scalability since the mean time of one MCMC iteration remains mostly constant for quite long intervals of threads equal to the number of galaxies.

VI. Conclusion

We described a practical methodology for the cost-effective parallelization of scientific applications, and we applied it to an astrophysical case, where we modelled the kinematic profiles of a sample of galaxies considered at the same time with a MCMC. The MCMC is a deterministic sequence dominated by a strict true dependence between successive steps, and therefore it cannot be fully parallelized. We defined a checkpoint every 1000 MCMC iterations, removing the need to restart the MCMC from the beginning when the code execution is interrupted. Within the MCMC for-loop, another internal for-loop iterates on an array of galaxies, which are independent of each other. We parallelized this loop with OpenMP. For reproducibility of results, the code and the input dataset have been made publicly available (https://github.com/alpha-unito/astroMP).

We analysed the performance of this code by investigating its strong and weak scaling. Regarding the former, results show that the maximum gain factor obtained by parallelizing this code is around 12 on a 48-core platform and around 8 on a 24-core platform. Concerning weak scalability, the mean time spent by one MCMC iteration is approximately constant in large intervals of threads equal in number to the used galaxies, where one galaxy is considered as our unit of work.

We pass from a total computing time of 6.1 Ms (∼71 days) for the sequential version of the code to a total computing time of 0.52 Ms (∼6 days) for the parallel version of the code with the maximum gain factor, considering all the 20000 MCMC iterations.

Given the simplicity and the generality of the proposed parallelization methodology, it can be applied to different problems concerning different topics, like particle simulations, optimization algorithms, ODE solvers or particle hydrodynamics simulations. In general, it can be applied to any code whose control flow statements are made of loops, which can be defined either in sequence or nested.

Acknowledgment

This paper is a revised version of the project work of the PhD students Valentina Cesare and Iacopo Colonnelli for the course *"Parallel and Distributed Computing System"* (MFN0795, taught by Marco Aldinucci) at the Computer Science Department in 2018-19. Experimentation has been supported by the Competency Center on Scientific Computing (C3S) at the University of Turin [30] and by the HPC4AI project funded by the Region Piedmont POR-FESR 2014-20 (INFRA-P) [31].

We sincerely thank Guy Tremblay for reading this paper and for his comments and suggestions which considerably improved the structure and the readibility of this work.

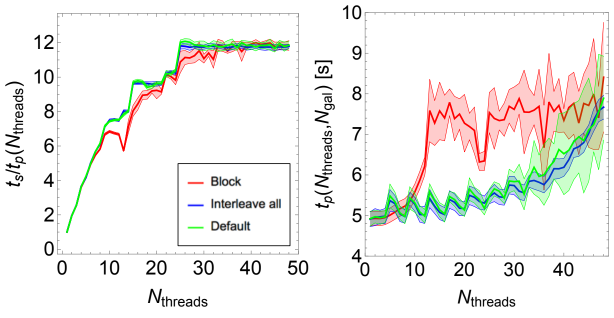

Figure 3. **Left:** strong scaling (Amdahl's law). **Right:** weak scaling (Gustafson's law). Both plots are referred to an Intel 48-core platform. The three solid lines (red, blue, green) refer to application run using different *NUMA control* policies for scheduling threads onto cores.

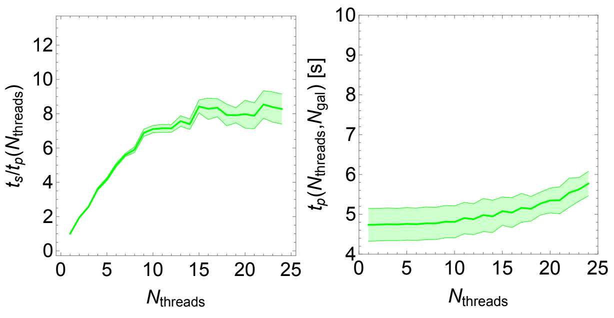

Figure 4. **Left:** strong scaling (Amdahl's law). **Right:** weak scaling (Gustafson's law). Both plots are referred to an Intel 24-core platform.

References

[1] M. Cole, "A skeletal approach to exploitation of parallelism," in *Proc. of CONPAR 88*, ser. British Computer Society Workshop Series. Cambridge University Press, 1989, pp. 667–675.

[2] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi, "P3L: a structured high level programming language and its structured support," *Concurrency Practice and Experience*, vol. 7, no. 3, pp. 225–255, May 1995.

[3] M. Aldinucci and M. Danelutto, "Algorithmic skeletons meeting grids," *Parallel Computing*, vol. 32, no. 7, pp. 449–462, 2006.

[4] J. Enmyren and 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*, ser. HLPP '10. New York, NY, USA: ACM, 2010, pp. 5–14.

[5] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," in *Programming Multi-core and Many-core Computing Systems*, ser. Parallel and Distributed Computing, S. Pllana and F. Xhafa, Eds. Wiley, 2017, ch. 13.

[6] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," *Softw., Pract. Exper.*, vol. 40, no. 12, pp. 1135–1160, 2010.

[7] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *Proc. of the 10th Intl. Conference on Autonomic Computing*. San Jose, CA: USENIX, 2013, pp. 23–27.

[8] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Usenix OSDI '04*, Dec. 2004, pp. 137–150.

[9] B. Pottenger and R. Eigenmann, "Idiom recognition in the Polaris parallelizing compiler," in *Proc. of the 9th Intl. Conference on Supercomputing (ICS '95)*. New York, NY, USA: ACM Press, 1995, pp. 444–448.

[10] M. Danelutto, J. D. García, L. M. Sánchez, R. Sotomayor, and 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.

[11] C. Lengauer, "Loop parallelization in the polytope model," in *Proc. of the 4th Intl. Conference on Concurrency Theory (CONCUR)*, ser. LNCS, vol. 715. Hildesheim, Germany: Springer, 1993, pp. 398–416.

[12] *OpenACC Directives for Accelerators*, Khronos Compute Working Group, Nov. 2012, http://www.openacc-standard.org.

[13] M. Aldinucci, M. Danelutto, M. Drocco, P. Kilpatrick, C. Misale, G. Peretti Pezzi, and M. Torquati, "A parallel pattern for iterative stencil + reduce," *Journal of Supercomputing*, vol. 74, no. 11, pp. 5690–5705, 2018.

[14] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proc. of the 15th ACM Symposium on Principles of Programming Languages (POPL)*. New York, NY, USA: ACM, 1988, pp. 319–329.

[15] Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke, "Uncovering hidden loop level parallelism in sequential applications," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, Feb 2008, pp. 290–301.

[16] C. Terboven, A. Spiegel, D. an Mey, S. Gross, and 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*, ser. LNCS, vol. 4315. Springer, 2005, pp. 95–106.

[17] J. Neal, T. Fewtrell, and M. Trigg, "Parallelisation of storage cell flood models using OpenMP," *Environmental Modelling & Software*, vol. 24, no. 7, pp. 872 – 877, 2009.

[18] V. Amaral *et al.*, "Programming languages for data-intensive HPC applications: A systematic mapping study," *Parallel Computing*, p. 102584, 2019.

[19] D. del Rio Astorga, M. F. Dolz, L. M. Sánchez, J. D. García, M. Danelutto, and M. Torquati, "Finding parallel patterns through static analysis in C++ applications," *IJHPCA*, vol. 32, no. 6, 2018.

[20] N. Wirth, *Algorithms and data structures*. Prentice Hall, 1985.

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

[22] A. J. Bernstein, "Analysis of programs for parallel processing," *IEEE Transactions on Electronic Computers*, vol. EC-15, no. 5, pp. 757–763, Oct 1966.

[23] J. Weidendorfer, M. Kowarschik, and C. Trinitis, "A tool suite for simulation based analysis of memory access behavior," in *Proc. of 4th Intl. Conference on Computational Science (ICCS)*, ser. LNCS, vol. 3038. Kraków, Poland: Springer, 2004, pp. 440–447.

[24] Y. Tanaka, K. Taura, M. Sato, and 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.

[25] 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*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485.

[26] J. L. Gustafson, "Reevaluating Amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988.

[27] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[28] M. A. Bershady, M. A. W. Verheijen, R. A. Swaters, D. R. Andersen, K. B. Westfall, and T. Martinsson, "The diskmass survey. i. overview," *The Astrophysical Journal*, vol. 716, no. 1, p. 198–233, May 2010.

[29] D. Young, "Iterative methods for solving partial difference equations of elliptic type," *Transactions of the American Mathematical Society*, vol. 76, no. 1, pp. 92–111, 1954.

[30] M. Aldinucci, S. Bagnasco, S. Lusso, P. Pasteris, and S. Rabellino, "Occam: a flexible, multi-purpose and extendable hpc cluster," in *Journal of Physics: Conf. Series (CHEP 2016)*, vol. 898, no. 8, San Francisco, USA, 2017, p. 082039.

[31] M. Aldinucci *et al.*, "HPC4AI, an AI-on-demand federated platform endeavour," in *ACM Computing Frontiers*, Ischia, Italy, May 2018.