DESY 18-089
SI-HEP-2018-32

Parallel Adaptive Monte Carlo Integration with the Event Generator WHIZARD

Simon Brass[1a], Wolfgang Kilian[2a,d], Jürgen Reuter[3b],

[a] *University of Siegen, Department of Physics, D–57068 Siegen, Germany,*
[b] *DESY Theory Group, D–22607 Hamburg, Germany*

Abstract

We describe a new parallel approach to the evaluation of phase space for Monte-Carlo event generation, implemented within the framework of the Whizard package. The program realizes a twofold self-adaptive multi-channel parameterization of phase space and makes use of the standard `OpenMP` and `MPI` protocols for parallelization. The modern `MPI3` feature of asynchronous communication is an essential ingredient of the computing model. Parallel numerical evaluation applies both to phase-space integration and to event generation, thus covering the most computing-intensive parts of physics simulation for a realistic collider environment.

[1] `simon.brass@uni-siegen.de`
[2] `kilian@physik.uni-siegen.de`
[3] `juergen.reuter@desy.de`

Contents

1 Introduction

Monte-Carlo event generators are an indispensable tool of elementary particle physics. Comparing collider data with a theoretical model is greatly facilitated if there is a sample of simulated events that represents the theoretical prediction, and can directly be compared to the event sample from the particle detector. The simulation requires two steps: the generation of particle-level events, resulting in a set of particle species and momentum four-vectors, and the simulation of detector response. To generate particle-level events, a generator computes partonic observables and partonic event samples which then are dressed by parton shower, hadronization, and hadronic decays. In this paper, we focus on the efficient computation of partonic observables and events.

Hard scattering processes involve Standard-Model (SM) elementary particles – quarks, gluons, leptons, $W^{\pm}$, Z, and Higgs bosons, and photons. The large number and complexity of scattering events recorded at detectors such as ATLAS or CMS, call for a matching computing power in simulation. Parallel evaluation that makes maximum use of available resources is an obvious solution.

The dominant elementary processes at the Large Hadron Collider (LHC) can be described as $2 \to 2$ or $2 \to 3$ particle production, where resonances in the final state subsequently decay, and additional jets can be accounted for by the parton shower. Cross sections and phase-space distributions are available as analytic expressions. Since distinct events are physically independent of each other, parallel evaluation is done trivially by generating independent event samples on separate processors. In such a situation, a parallel simulation run on a multi-core or multi-processor system can operate close to optimal efficiency.

However, the LHC does probe rarer partonic processes which are of the type $2 \to n$ where $n \geq 4$. There are increasing demands on the precision in data analysis at the LHC and, furthermore, at the planned future high-energy and high-luminosity lepton and hadron colliders. This forces the simulation to go beyond the leading order in perturbation theory, beyond the separation of production and decay, and beyond simple approximations for radiation. For instance, in processes such as top-quark pair production or vector-boson scattering, the simulation must handle elementary $n = 6, 8, 10$ processes. Closed analytical expressions for arbitrary phase-space distributions are not available.

Event generators for particle physics processes therefore rely on statistical methods both for event generation and for the preceding time-consuming numerical integration of the phase space. All major codes involve a Monte-Carlo rejection algorithm for generating unweighted event samples. This requires knowledge of the total cross section and of a reference limiting distribution over the whole phase space. Calculating those quantities relies again on Monte-Carlo methods, which typically involve an adaptive

iteration algorithm. A large fraction of the total computing time cannot be trivially parallelized since it involves significant communication. Nevertheless, some of the main MC event generators have implemented certain parallelization features, e.g., `Sherpa` [1] and `MG5_aMC@NLO` [2].

In the current paper, we describe a new approach to efficient parallelization for adaptive Monte Carlo integration and event generation, implemented within the WHIZARD [3] Monte-Carlo integration and event-generation program. The approach combines independent evaluation on separate processing units with asynchronous communication via MPI 3.0 with internally parallelized loops distributed on multiple cores via OpenMP. In Sec. 2, we give an overview of the workflow of the WHIZARD event generation framework with the computing-intensive tasks that it has to perform. Sec. 3 describes the actual MC integration and event generation algorithm. The parallelization methods and necessary modifications to the algorithm are detailed in Sec. 4. This section also shows our study on the achievable gain in efficiency for typical applications in high-energy physics. Finally, we conclude in Sec. 5.

2 The WHIZARD multi-purpose event generator framework

We will demonstrate our parallelization algorithms within the WHIZARD framework [3]. WHIZARD is a multi-purpose Monte-Carlo integration and event generator program. In this Section, we describe the computing-intensive algorithms and tasks which are potential targets of improvement via parallel evaluation. In order to make the section self-contained, we also give an overview of the capabilities of WHIZARD.

The program covers the complete workflow of particle-physics calculations, from setting up a model Lagrangian to generating unweighted hadronic event samples. To this end, it combines internal algorithms with external packages. Physics models are available either as internal model descriptions or via interfaces to external packages, e.g. for FEYNRULES [4]. For any model, scattering amplitudes are automatically constructed and accessed for numerical evaluation via the included matrix-element generator O'MEGA [5, 6, 7, 8, 9, 10]. The calculation of partonic cross sections, observables, and events is handled within the program itself, as detailed below. Generated events are showered by internal routines [11], showered and hadronized by standard interfaces to external programs, or by means of a dedicated interface to PYTHIA [12, 13]. For next-to-leading-order (NLO) calculations, WHIZARD takes virtual and color-/charge-/spin-correlated matrix elements from the one-loop providers OPENLOOPS [14, 15], GOSAM [16, 17], or RECOLA [18], and handles subtraction within the Frixione-Kunszt-Signer scheme (FKS) [19, 20, 21, 22]. Selected WHIZARD results at NLO, some of them obtained using parallel evaluation as presented in this paper, can be found in Refs. [23, 24, 25, 26, 27].

The core part of WHIZARD is the phase-space integration and its parameterization in terms of importance-ordered channels. The phase space is the manifold of kinematically allowed energy-momentum configurations of the external particles in an elementary process. User-defined cuts may additionally constrain phase space in rather arbitrary ways. WHIZARD specifically allows for arbitrary phase-space cuts, that can be steered from the input file via its scripting language SINDARIN without any (re-)compilation of code. The program determines a set of phase-space parameterizations (called *channels*), i.e., bijective mappings of a subset of the unit d-dimensional hypercube onto the phase-space manifold. For the processes of interest, d lies between 2 and some 25 dimensions. Note that the parameterization of non-trivial beam structure in form of parton distribution functions, beam spectra, electron structure functions for initial-state radiation, effective photon approximation, etc., provides additional dimensions to the numerical integration. The actual integrand, i.e., the square of a transition matrix element evaluated at the required order in perturbation theory, is defined as a function on this cut phase space. It typically contains sharp peaks (resonances) and develops poles in the integration variables just beyond the boundaries. We collectively denoted those as "singularities" in a slight abuse of language. In effect, the numerical value of the integrand varies over many orders of magnitude.

For an efficient integration, it is essential that the program generates multiple phase-space channels for the same process. Each channel has the property that a particular subset of the singularities of the integrand maps to a slowly varying distribution (in the ideal case a constant) along a coordinate axis of the phase-space manifold with that specific mapping. The set of channels has to be large enough that it covers all numerically relevant singularities. This is not a well-defined requirement, and WHIZARD contains a heuristic algorithm that determines this set. The number of channels may range from one (e.g. $e^+e^- \to \mu^+\mu^-$ at $\sqrt{s} = 40$ GeV, no beam structure) to some 10^6 (e.g. vector boson scattering at the LHC, or BSM processes at the LHC) for typical applications.

The actual integration is done by the VAMP subprogram, which is a multi-channel version of the self-adaptive VEGAS algorithm. We describe the details of this algorithm below in Sec. 3. In essence, for

each channel, the hypercube is binned along each integration dimension, and the bin widths as well as the channel weight factors are then iteratively adapted in order to reduce the variance of the integrand as far as possible. This adaptive integration step includes the most computing-intensive parts of WHIZARD. CPU time is spent mostly in (i) the evaluation of the matrix element at each phase space point which becomes particularly time-consuming for high-multiplicity or NLO processes, (ii) the evaluation of the phase-space mapping and its inverse for each channel, alongside with the Jacobian of this mapping, (iii) sampling the phase-space points and collecting the evaluation results, and (iv) the adaption of the bin widths and channel weights. If the adaptation is successful, it will improve the integration result: the relative error of the Monte-Carlo integration is estimated as

$$\frac{\Delta I}{I} = \frac{a}{\sqrt{N}}, \tag{1}$$

where I is the integral that has to be computed, ΔI is the statistical error of the integral estimate, and N is the number of phase-space points for which the integrand has been evaluated. Successful adaptation will improve the integral error by reducing the *accuracy* parameter a, often by several orders of magnitude.

The program records the integration result. The accompanying set of adapted bin widths, for each dimension and for each channel, together with the maximum value of the mapped integrand, is called an integration *grid*, and is also recorded. The set of adapted grids can then be used for the final simulation step. Simulation implies generating a sample of statistically independent events (phase-space points), with a probability distribution as given by the integrand value. After multiple adaptive iterations, the effective event weights should vary much less in magnitude compared to the original integrand defined on the phase-space manifold. Assuming that the maximum value can be estimated sufficiently well, a simple rejection algorithm can convert this into a set of unweighted events. An unweighted event sample constitutes an actual simulation of an experimental outcome. The *efficiency* ϵ of this reweighting is the ratio of the average event weight over the maximum weight,

$$\epsilon = \frac{\langle w \rangle_N}{w_{\max}}. \tag{2}$$

Since the integrand has to be completely evaluated for each event before acception or rejection, the unweighting efficiency ϵ translates directly into CPU time for generating an unweighted event sample. Successful adaptation should increase ϵ as far as possible. We note that event generation, for each phase-space point, involves all channel mappings and Jacobian calculations in the same way as integration does.

Finally, for completeness, we note that WHIZARD contains additional modules that implement other relevant physical effects, e.g., incoming-beam structure, polarization, factorizing processes into production and decay, and modules that prepare events for actual physics studies and analyses. To convert partonic events into hadronic events, the program provides its own algorithms together with, or as an alternative to, external programs such as PYTHIA. Data visualization and analysis can be performed by its own routines or by externally operating on event samples, available in various formats.

Before we discuss the parallelization of the phase-space integration, in the next section, Sec. 3, we explain in detail how the MC integration of WHIZARD works.

3 The MC integrator of WHIZARD: the VAMP algorithm

The implementation of the integration and event generation modules of WHIZARD is based on the VEGAS algorithm [28, 29]. WHIZARD combines the VEGAS method of adaptive Monte-Carlo integration with the principle of multi-channel integration [30]. The basic algorithm and a sample implementation have been published as VAMP (Vegas AMPlified) in [31]. In this section, in order to make the discussion of our parallelized re-implementation of the VAMP algorithm, we discuss in detail the algorithm and its application to phase-space sampling within WHIZARD. Our parallelized implementation for the purpose of efficient parallel evaluation is then presented in Sec. 4.

3.1 Integration by Monte-Carlo Sampling

We want to compute the integral I for an integrand f defined on a compact d-dimensional phase-space manifold Ω. The integrand f represents a real-valued squared matrix element (in NLO calculations, this can be a generalization that need not be positive semidefinite) with potentially high numerical variance.

The coordinates p represent four-momenta and, optionally, extra integration variables such as structure function parameters like parton energy fractions. The phase-space manifold and the measure $\mathrm{d}\mu(p)$ are determined by four-momentum conservation, on-shell conditions, and optionally by user-defined cuts and weight factors:

$$I_\Omega[f] = \int_\Omega \mathrm{d}\mu(p)\, f(p)\,. \tag{3}$$

A phase-space parameterization is a bijective mapping ϕ from a subset U of the d-dimensional unit hypercube, $U \subset (0,1)^d$, onto $\Omega = \phi(U)$ with Jacobian $\phi' = \det(\mathrm{d}\phi/\mathrm{d}x)$,

$$p = \phi(x), \qquad \mathrm{d}\mu(p) = \phi'(x)\,\mathrm{d}\mu(x) = \phi'(x)\,\rho_\phi(x)\,\mathrm{d}^d x\,. \tag{4}$$

The measure in this chart becomes proportional to the canonical measure on $\mathbb{R}^d$ with density $\phi'(x)\,\rho_\phi(x)$. Mappings ϕ may be chosen a priori such that the density expressed in these coordinates does not exhibit high numerical variance; for instance, the Jacobian ϕ' may cancel integrable singularities associated with Gram determinants near the edges of phase space. Furthermore, we may extend the x integration over the complete unit hypercube, continuing Ω arbitrarily while setting $\mathrm{d}\mu(x) = 0$ for values x outside the integration domain U (e.g. outside a fiducial phase-space volume).

The basic idea of Monte-Carlo integration [32] builds upon the observation that the integral $I_\Omega[f]$ can be estimated by a finite sum, where the estimate is given by

$$E_N[f] = \langle f \rangle_N = \frac{1}{N} \sum_{i=1}^{N} f(\phi(x_i))\,\phi'(x_i)\,\rho_\phi(x_i), \tag{5}$$

if the points x_i are distributed according to a uniform random distribution within the hypercube. N is the number of random number configurations for which the integrand has been evaluated, the *calls*. The integration method is also known as *importance sampling*.

Asymptotically, the estimators $E_N[f]$ for independent random sequences $\{x_i\}$ will themselves be statistically distributed according to a Gaussian around the mean value $I_\Omega[f]$. The statistical error of the estimate can be given as the square root of the variance estimate,

$$V_N[f] = \frac{N}{N-1}\left(\langle f^2 \rangle_N - \langle f \rangle_N^2\right), \tag{6}$$

which is calculated alongside with the integral estimate. Asymptotically, the statistical error of the integration scales according to Eq. (1), where the accuracy a depends on the actual variance of the effective integrand

$$f_\phi(x) = f(\phi(x))\,\phi'(x)\,\rho_\phi(x) \tag{7}$$

In short, the computation consists of a sequence of *events*, points x_i with associated weights $w_i = f_\phi(x_i)$.

This algorithm has become a standard choice for particle-physics computations, because (i) the error scaling law $\propto 1/\sqrt{N}$ turns out to be superior to any other useful algorithm for large dimensionality of the integral d; (ii) by projecting on observables $\mathcal{O}(\phi(x))$, any integrated or differential observable can be evaluated from the same event sample; and (iii) an event sample can be unweighted to accurately simulate the event sample resulting from an actual experiment. The unweighting efficiency ϵ as in Eq. (2) again depends on the behavior of the effective integrand.

The optimal values $a = 0$ and $\epsilon = 1$ are reached if $f(\phi(x))\,\phi'(x)\,\rho_\phi(x) \equiv 1$. In one dimension, this is possible by adjusting the mapping $\phi(x)$ accordingly. The Jacobian ϕ' should cancel the variance of the integrand f, and thus will assume the shape of this function. In more than one dimension, such mappings are not available in closed form, in general.

In calculations in particle-physics perturbation theory, the integrand f is most efficiently derived recursively, e.g. from on-particle off-shell wavefunctions like in [5]. The poles in these recursive structures are the resonant Feynman propagators. In particular, if for a simple process only a single propagator contributes, there are standard mappings ϕ such that the mapped integrand factorizes into one-dimensional functions, and the dominant singularities are canceled. For this reason, the phase space channels of WHIZARD are constructed from dominating proagators, which are represented by typical Feynman graphs (even if the matrix elements do not rely on the redundant expansion into Feynman graphs). If several graphs contribute, mappings that cancel the singularities are available only for very specific cases such as massless QCD radiation, e.g. [33, 34]. In any case, we have to deal with some remainder variance that is not accounted for by standard mappings, such as polynomial factors in the numerator, higher-order contributions, or user-defined cuts which do not depend on the integrand f.

3.2 The VEGAS algorithm: importance sampling

The VEGAS algorithm [29] addresses the frequent situation that the effective integrand f_ϕ is not far from factorizable form, but the capabilities of finding an optimal mapping ϕ in closed form have been exhausted. In that case, it is possible to construct a factorizable *step* mapping that improves accuracy and efficiency beyond the chosen $\phi(x)$.

Several implementations of VEGAS exist, e.g. Lepage's `FORTRAN77` implementation [29] or the GNU scientific library C implementation [35]. Here, we relate to the VAMP integration package as it is contained in WHIZARD. It provides an independent implementation which realizes the same basic algorithm and combines it with multi-channel integration, as explained below in Sec. 3.4.

Let us express the integration variables x in terms of another set of variables r, defined on the same unit hypercube. The mapping $r = G(x)$ is assumed bijective, factorizable, and depends on a finite set of adjustable parameters. If r_i are uniformly distributed random numbers, the distribution of $x_i = G^{-1}(r_i)$ becomes non-uniform, and we have to compensate for this by dividing by the Jacobian $g(x) = G'(x)$,

$$I_\Omega[f] = \int_U f_\phi(x)\,\mathrm{d}^d x = \int_{G(U)=U} \left.\frac{f_\phi(x)}{g(x)}\right|_{x=G^{-1}(r)} \mathrm{d}^d r\,. \tag{8}$$

Alternatively, we may interpret this result as the average of f_ϕ/g, sampled with an x distribution that follows the probability density $g(x)$, cf. [31]:

$$I_\Omega[f] = \left\langle \frac{f}{g} \right\rangle_g \tag{9}$$

For a finite sample with N events, the estimators for integral and variance are now given by

$$E_N(I) = \frac{1}{N}\sum_{i=1}^{N} \frac{f_\phi(x_i)}{g(x_i)}, \tag{10}$$

$$V_N(I) = \frac{N}{N-1}\left(\frac{1}{N}\sum_{i=1}^{N}\left(\frac{f_\phi(x_i)}{g(x_i)}\right)^2 - E_N(I)^2\right), \tag{11}$$

where the x_i are computed from the uniformly distributed r_i via $x_i = G^{-1}(r_i)$.

The VEGAS algorithm makes the following particular choice for the mapping G (or its derivative $g = G'$): For each dimension $k = 1, \ldots d$, the interval $(0, 1)$ is divided into n_k bins B_{kj_k} with bin width Δx_{kj_k}, $j_k = 1, \ldots n_k$, such that the one-dimensional probability distribution $g_k(x_k)$ is constant over that bin B_{kj_k} and equal to its inverse bin width, $1/\Delta x_{kj_k}$. The overall probability distribution $g(x)$ is also constant within each bin and given by

$$g(x) = \prod_k g_k(x_k) = \prod_k \frac{1}{n_k \Delta x_{kj_k}}, \tag{12}$$

if $x_k \in B_{kj_k}$. It is positive definite and satisfies

$$\int_U g(x)\,\mathrm{d}^d x = 1 \tag{13}$$

by construction, and thus defines an acceptable mapping G.

The iterative adaptation algorithm starts from equidistant bins. It consists of a sequence of integration passes, where after each pass, the bin widths are adapted based on the variance distribution within that pass [28, 29]. The adaptive mechanism of VEGAS adjusts the size of the bins for each bin j_k of each axis k based on the size of the following measure:

$$m_{j_k} = \left[\left(\frac{\omega_{j_k}}{\sum \omega_{j_k}} - 1\right)\frac{1}{\log \frac{\omega_{j_k}}{\sum \omega_{j_k}}}\right]^\alpha \qquad \begin{cases} > 1 & \text{increase bin size} \\ = 1 & \text{keep bin size} \\ < 1 & \text{decrease bin size} \end{cases}, \tag{14}$$

respecting the overall normalization. The power α is a free parameter. The individual (squared) bin weights in Eq. (14) are defined as

$$\omega_{j_k}^2 = (\langle f_i \rangle \Delta x_i)^2 = \sum_{\substack{i \\ (x_i)_k \in B_{kj_k}}} \frac{f^2(x_i)}{g^2(x_i)}\,\rho(x_i). \tag{15}$$

i.e., all integration dimensions $k' \neq k$ are averaged over when adjusting the bins along dimension k. These bin weights are easily accumulated while sampling events for the current integration pass.

This is an optimization problem with $n = \sum_{k=1}^{d}(n_k - 1)$ free parameters, together with a specific strategy for optimization. If successful, the numerical variance of the ratio $f_\phi(x)/g(x)$ is reduced after each adaptation of g. In fact, the shape of $g(x)$ will eventually resemble a histogrammed version of $f_\phi(x)$, with a saw-like profile along each integration dimension. Bins will narrow along slopes of high variation in f_ϕ, such that the ratio f_ϕ/g becomes bounded. The existence of such a bound is essential for unweighting events, since the unweighting efficiency ϵ scales with the absolute maximum of $f_\phi(x)/g(x)$ within the integration domain. Clearly, the value of this maximum can only be determined with some uncertainty since it relies on the finite sample $\{x_i\}$. The saw-like shape puts further limits on the achievable efficiency ϵ. Roughly speaking, each direction with significant variation in f_ϕ reduces ϵ by a factor of two.

The set of updated parameters Δx_{kj_k} defines the integration grid for the next iteration. In the particle-physics applications covered by WHIZARD we have $d \lesssim 30$, the number of bins is typically chosen as $n_k \lesssim 30$, all n_k equal, so a single grid consists of between a few and 10^3 parameters n subject to adaptation. In practice, the optimization strategy turns out to be rather successful. Adapting the grid a few times does actually improve the accuracy a and the efficiency ϵ significantly. Only the grids from later passes are used for calculating observables and for event generation. Clearly, the achievable results are limited by the degree of factorizability of the integrand.

3.3 The VEGAS algorithm: (pseudo-)stratified sampling

The importance-sampling method guarantees, for a fixed grid, that the estimator E_N for an integrand approaches the exact integral for $N \to \infty$. Likewise, a simulated unweighted event sample statistically approaches an actual observed event sample, if the integrand represents the actual matrix element.

However, the statistical distribution of the numbers x_i is a rather poor choice for an accurate estimate of the integral. In fact, in one dimension a simple equidistant bin-midpoint choice for x_i typically provides much better convergence than $1/\sqrt{N}$ for the random distribution. A reason for nevertheless choosing the Monte-Carlo algorithm is the fact that for n bins in d dimensions, the total number of cells is n^d, which easily exceeds realistic values for N: for instance, $n = 20$ and $d = 10$ would imply $n^d = 10^{13}$, but evaluating the integrand at much more than 10^7 points may already become infeasible.

The *stratified sampling* approach aims at combining the advantages of both methods. Binning along all coordinate axes produces n^d cells. Within each cell, the integrand is evaluated at precisely s distinct points, $s \geq 2$. We may choose n such that the total number of calls, $N = s \cdot n^d$, stays within limits feasible for a realistic sampling. For instance, for $s = 2$, $d = 10$, and limiting the number of calls to $N \approx 10^7$, we obtain $n = 4 \ldots 5$. Within each cell, the points are randomly chosen, according to a uniform distribution. Again, the VEGAS algorithm iteratively adapts the binning in several passes, and thus improves the final accuracy.

For the problems addressed by WHIZARD, pure stratified sampling is not necessarily an optimal approach. The structure of typical integrands cannot be approximated well by the probability distribution $g(x)$ if the number of bins per dimension n is small. To allow for larger n despite the finite total number of calls, the *pseudo-stratified* approach applies stratification not in x space, which is binned into n_x^d cells with $n \lesssim 20$, but in r space which was not binned originally. The n_r bins in r space are not adapted, so this distribution stays uniform. In essence, the algorithm scans over all n_r^d cells in r space and selects two points randomly within each r cell, and then maps those points to points in x space, where they end up in any of the n_x^d cells. The overall probability distribution in x is still $g(x)$ as given by Eq. (12), but the distribution has reduced randomness in it and thus yields a more accurate integral estimate.

Regardless of the integration algorithm, simulation of unweighted events can only proceed via strict importance sampling. Quantum mechanics dictates that events have to be distributed statistically independent of each other over the complete phase space. Therefore, WHIZARD separates its workflow into integration passes which adapt integration grids and evaluate the integral, and a subsequent simulation run which produces an event sample. The integration passes may use either method, while event generation uses importance sampling and, optionally, unweighting the generated events. In practice, using grids which have been optimized by stratified sampling is no disadvantage for subsequent importance sampling since both sampling methods lead to similarly shaped grids.

3.4 Multi-channel integration

The adaptive Monte-Carlo integration algorithms described above do not yield satisfactory results if the effective integrand f_ϕ fails to factorize for the phase-space channel ϕ. In non-trivial particle-physics processes, many different Feynman graphs, possibly with narrow resonances, including mutual interference, contribute to the integrand.

Ref. [30] introduced a multi-channel ansatz for integration that ameliorates this problem. The basic idea is to introduce a set of K different phase-space channels $\phi_c : U \to \Omega$, corresponding coordinates x_c with $p = \phi_c(x_c)$ and densities $\rho_\phi(x_c)$ with $\mathrm{d}\mu(p) = \phi_c'(x_c)\,\rho_\phi(x_c)\,\mathrm{d}^d x_c$, and a corresponding set of *channel weights* $\alpha_c \in \mathbb{R}$ which satisfy

$$0 \le \alpha_c \le 1, \qquad \sum_{c=1}^{K} \alpha_c = 1 \quad . \tag{16}$$

We introduce the function

$$h(p) = \sum_c \alpha_c \frac{1}{\phi_c'(\phi_c^{-1}(p))} \quad , \tag{17}$$

which depends on the Jacobians ϕ_c' of all channels. Using this, we construct a partition of unity,

$$1 = \sum_c \alpha_c \frac{1}{\phi_c'(\phi_c^{-1}(p))\,h(p)} \quad , \tag{18}$$

which smoothly separates phase space into regions where the singularities dominate that are mapped out by any individual channel ϕ_c, respectively.

The master formula for multi-channel integration makes use of this partition of unity and applies, for each term, its associated channel mapping ϕ_c.

$$I_\Omega[f] = \int_\Omega f(p)\,\mathrm{d}\mu(p) = \int_\Omega \sum_c \alpha_c \frac{f(p)}{h(p)}\,\frac{\mathrm{d}\mu(p)}{\phi_c'} \tag{19}$$

$$= \sum_c \alpha_c \int_U \frac{f(\phi_c(x_c))}{h(\phi_c(x_c))}\,\rho_{\phi_c}(x_c)\,\mathrm{d}^d x_c \quad . \tag{20}$$

The mappings ϕ_c are chosen such that any singularity of f is canceled by at least one of the Jacobians ϕ_c'. In the vicinity of this singularity, ϕ_c' approaches zero in Eq. (17), and the effective integrand

$$f_c^h(x_c) = \frac{f(\phi_c(x_c))}{h(\phi_c(x_c))}\,\rho_{\phi_c}(x_c), \tag{21}$$

becomes

$$f_c^h(x_c) \sim \frac{1}{\alpha_c} f_{\phi_c}(x_c) \quad . \tag{22}$$

We thus benefit from the virtues of phase-space mapping in the original single-channel version, but cancel all singularities at once. Each effective integrand f_c^h which depends on all weights α_c and Jacobians ϕ_c' simultaneously, is to be integrated in its associated phase space channel. The results are added, each integral weighted by α_c.

The importance sampling method can then be applied as before,

$$E_N[f] = \sum_c \alpha_c \frac{1}{N_c} \sum_{i_c=1}^{N_c} f_c^h(x_{ci_c}), \tag{23}$$

where the total number of events N is to be distributed among the integration channels, $N = \sum_c N_c$. A possible division is to choose α_c such that $N_c = \alpha_c N$ is an integer for each channel, and thus

$$E_N[f] = \frac{1}{N} \sum_c \sum_{i_c=1}^{N_c} f_c^h(x_{ci_c}) \tag{24}$$

becomes a simple sum where the integration channels are switched according to their respective weights. Within each channel, the points x_{ci_c} can be taken as uniformly distributed random numbers. Alternatively, we may apply stratified sampling as before, within each channel.

The weights α_c are free parameters, and thus an obvious candidate for optimization. We may start from a uniform distribution of weights among channels, $\alpha_c = 1/K$, and adapt the weights iteratively. In analogy to the VEGAS rebinning algorithm, we accumulate the total variance for each channel c to serve as a number ω_c which enters an update formula analogous to Eq. (15), with an independent power β (cf. Eq. (24) in Ref. [31], and Ref. [30]):

$$\alpha_c \to \frac{\alpha_c \omega_c^\beta}{\sum_c \alpha_c \omega_c^\beta} \quad . \tag{25}$$

This results in updated weights α_c. The weights, and the total number of events N for the next iteration, are further adjusted slightly such that the event numbers N_c become again integer. Furthermore, it may be useful to insert safeguards for channels which by this algorithm would acquire very low numbers N_c, causing irregular statistical fluctuations. In that case, we may choose to either switch off such a channel, $\alpha_c = 0$, or keep N_c at some lower threshold value, say $N_c = 10$. These refinements are part of the WHIZARD setup.

Regarding particle-physics applications, a straightforward translation of (archetypical representatives of) Feynman graphs into integration channels can result in large values for the number of channels K, of order 10^5 or more. In fact, if the number of channels increases proportional to the number of Feynman graphs, it scales factorially with the number of elementary particles in the process. This is to be confronted with the complexity of the transition-matrix calculation, where recursive evaluation results in a power law. Applied naively, multi-channel phase-space sampling can consume the dominant fraction of computing time. Furthermore, if the multi-channel approach is combined with adaptive binning (see below), the number of channels is multiplied by the number of grid parameters, so the total number of parameters grows even more quickly. For these reasons, WHIZARD contains a heuristic algorithm that selects a smaller set of presumably dominant channels for the multi-channel integration. Since all parameterizations are asymptotically equivalent to each other regarding importance sampling, any such choice does not affect the limit $E_N[f] \to I_\Omega[f]$. It does affect the variance and can thus speed up – or slow down – the convergence of the integral estimates for $N \to \infty$ and for iterative weight adaptation.

3.5 Doubly adaptive multi-channel integration: VAMP

The VAMP algorithm combines multi-channel integration with channel mappings ϕ_c with the VEGAS algorithm. For each channel $c = 1, \ldots K$, we introduce a bijective step mapping G_c of the unit hypercube onto itself $U \to G_c(U) = U$. The Jacobian is $g_c(x) = G_c'(x)$, where g_c factorizes along the coordinate axes (labeled by $k = 1, \ldots d$) and is constant within bins B_{c,kj_k} (labeled by $j_k = 1, \ldots n_{c,k}$),

$$g_c(x) = \prod_k g_{c,k}(x_{c,k}) = \prod_k \frac{1}{n_{c,k} \Delta x_{c,kj_k}} \ , \tag{26}$$

if $x_{c,k} \in B_{c,kj_k}$. The normalization condition

$$\int_U g_c(x_c)\, \mathrm{d}^d x_c = 1 \tag{27}$$

is satisfied for all channels, and enables us to construct G_c.

We chain the mappings G_c with the channel mappings ϕ_c in the partion of unity, Eq. (18), and write

$$I_\Omega[f] = \sum_c \alpha_c \int_U f_c^g(x_c) \Big|_{x_c = G_c^{-1}(r_c)} \mathrm{d}^d r_c \quad , \tag{28}$$

where the modified effective integrand for channel c is given by

$$f_c^g(x_c) = \frac{f(\phi_c(x_c))}{g(\phi_c(x_c))} \rho_{\phi_c}(x_c) \quad . \tag{29}$$

Here, $g(p)$ replaces $h(p)$ in (17),

$$g(p) = \sum_c \alpha_c \frac{g_c(x_c)}{\phi_c'(x_c)} \Big|_{x_c = \phi_c(p)} \quad . \tag{30}$$

The variance of the integrand is reduced not just by the fixed Jacobian functions ϕ'_c, but also by the tunable distributions g_c. In a region where one of the g_c distributions becomes numerically dominant, $\alpha_c f_c^g(x_c)$ approaches the single-channel expression $f_{\phi_c}(x_c)/g_c(x_c)$, cf. Eq. (8).

The multi-channel sampling algorithm can be expressed in form of the integral estimate $E_N[f]$,

$$E_N[f] = \frac{1}{N} \sum_c \sum_{i_c=1}^{N_c} f_c^g(x_{ci_c}) \qquad \text{with} \qquad x_{ci_c} = G_c^{-1}(r_{ci_c}), \tag{31}$$

and the r_{ci_c} are determined either from a uniform probability distribution in the unit hypercube, or alternatively, from a uniform probability distribution within each cell of a super-imposed stratification grid. The free parameters of this formula are α_c, $c = 1, \ldots K$, and for each channel, the respective channel grid with parameters $\Delta x_{c,k,j_k}$.

In order to improve the adaptation itself, similarity mappings between different channels can be used in order to achieve a better adaptation of the individual grids, because the distribution for grid adaptation is better filled. This in turn leads to an improved convergence of the integration and a better weighting efficiency of the event generation. For this, we define equivalences for channels that share a common structure. These equivalences allow adaptation information, i.e. the individual bin weights w_{j_k} of each axis, to be averaged over several channels, which in turn improves the statistics of the adaptation. Such an equivalence maps the individual bin weights of a channel c' onto the current channel c together with the permutation of the integration dimensions d, $\pi : c \mapsto c' : k \mapsto \pi(k)$, and the type of mapping. The different mappings used in that algorithm are:

$$\begin{array}{ll}
\textbf{identity} & w^c_{j_k} \to w^c_{j_k} + w^{c'}_{j_{\pi(k)}} \\
\textbf{invert} & w^c_{j_k} \to w^c_{j_k} + w^{c'}_{d-j_{\pi(k)}} \\
\textbf{symmetric} & w^c_{j_k} \to w^c_{j_k} + \tfrac{1}{2}(w^{c'}_{j_{\pi(k)}} + w^{c'}_{d-j_{\pi(k)}}) \\
\textbf{invariant} & w^c_{j_k} \to 1
\end{array} \tag{32}$$

Interesting applications are those that require very high statistics, as shown in the table 1. Channel equivalences have been shown to play a crucial role in sampling correctly particularly the less densely populated regions of phase space, e.g. in vector boson scattering [36, 37, 38, 39, 40, 41, 42] or in beyond the Standard Model (BSM) simulations with a huge number of phase space channels [8, 43, 44, 45]. Channel equivalences are used both for the traditional VAMP Monte Carlo integrator as well as for the new parallelized version. Constructing these channel equivalences is part of the phase space algorithm, but a detailed explanation is out of scope of this work.

Process	$\sigma_{\text{new}}\,1\,\text{fb}^{-1}$	$\sigma_{\text{original}}\,1\,\text{fb}^{-1}$
$\text{gg} \to \text{W}\,q\bar{q}$	$24\,097 \pm 22$	$24\,091 \pm 12$
$\text{gg} \to \text{W}\,q\bar{q}\text{g}$	9111 ± 15	9142 ± 6
$\text{gg} \to \text{W}\,q\bar{q}\text{gg}$	2310 ± 7	2363 ± 7
$\text{gg} \to \text{W}\,q\bar{q}\text{ggg}$	410 ± 130	523 ± 6
$jj \to \text{W}j$	$936\,600 \pm 6$	$936\,000 \pm 28$
$jj \to \text{W}jj$	$287\,490 \pm 26$	$287\,000 \pm 14$
$jj \to \text{W}jjj$	$79\,570 \pm 15$	$80\,000 \pm 6$
$jj \to \text{W}jjjj$	$17\,900 \pm 11$	$20\,900 \pm 6$

Table 1: *We show the numerical results for the cross sections of W + jets processes computed with the original* VAMP *and the reimplemented version. Both process and cut definitions have been taken from [3], as well as the original results of* VAMP*. The original results were calculated with equivalences and the results of the new implementation without equivalences. By comparison, the results are statistically consistent, but the results without equivalences are less accurate.*

The actual integration algorithm is organized as follows. Initially, all channel weights and bin widths are set equal to each other. There is a sequence of iterations where each step consists of first generating a sample of N events, then adapting the free parameters. This adaptation may update either the channel weights via Eq. (25) or the grids via Eq. (14), or both, depending on user settings. The event sample is divided among the selected channels based on event numbers N_c. For each channel, the integration hypercube in r is scanned by cells in terms of stratified sampling, or sampled uniformly (importance

sampling). For each point r_c, we compute the mapped point x_c, the distribution value $g_c(x_c)$, and the phase-space density $\rho_c(x_c)$ at this point. Given the fixed mapping ϕ_c, we compute the phase-space point p and the Jacobian factor ϕ'_c. This allows us to evaluate the integrand $f(p)$. Using p, we scan over all other channels $c' \neq c$ and invert the mappings to obtain $\phi'_{c'}$, $x_{c'}$, $g_{c'}(x_{c'})$, and $\rho_{c'}(x_{c'})$. Combining everything, we arrive at the effective weight $w = f^g_c(x_c)$ for this event. Accumulating events and evaluating mean, variance, and other quantities then proceeds as usual. Finally, we may combine one or more final iterations to obtain the best estimate for the integral, together with the corresponding error estimate.

If an (optionally unweighted) event sample is requested, WHIZARD will take the last grid from the iterations and sample further events, using the same multi-channel formulas, with fixed parameters, but reverting to importance sampling over the complete phase space. The channel selection is then randomized over the channel weights α_c, allowing for an arbitrary number of simulated physical events.

4 Parallelization of the WHIZARD workflow

In this section we discuss the parallelization of the WHIZARD integration and event generation algorithms. We start with a short definition of observables and timings that allow to quantify the gain of a parallelization algorithm in Sec. 4.1. Then, in Sec. 4.2, we discuss the computing tasks for a typical integration and event generation run with the WHIZARD program, while in Sec. 4.3 we list possible computing frameworks for our parallelization tasks and what we chose to implement in WHIZARD. Random numbers have to be set up with great care for parallel computations, as we point out in Sec. 4.4. The WHIZARD algorithm for parallelized integration and event generation is presented in all details in Sec. 4.5. Finally, in Sec. 4.6, we introduce an alternativ method to generate the phase-space parameterization that is more efficient for higher final-state particle multiplicities and is better suited for parallelization.

4.1 Basics

The time that is required for a certain computing task can be reduced by employing not a single processing unit (*worker*), but several workers which are capable of performing calculations independent of each other. In a slightly simplified view, we may assume that a bare program consists of parts that are performed by a single worker (time T_s), and of other parts that are performed by n workers simultaneously (time T_m). The serial time T_s also covers code that is executed identically on all workers. The total computing time can then be written as

$$T(n) = T_s + \frac{1}{n} T_m + T_c(n). \tag{33}$$

The extra time T_c denotes the time required for communication between the workers, and for workers being blocked by waiting conditions. Its dependence on n varies with the used algorithm, but we expect a function that vanishes for $n = 1$ and monotonically increases with n, e.g., $T_c \sim \log(n)$ or $T_c \sim (n-1)^\alpha$, $\alpha > 0$. The *speedup* factor of parallelization then takes the form

$$f(n) = \frac{T(1)}{T(n)} = \frac{T_s + T_m}{T_s + \frac{1}{n} T_m + T_c(n)} \quad , \tag{34}$$

and we want this quantity to become as large as possible.

Ideally, T_s and T_c vanish, and

$$f(n) = n \quad , \tag{35}$$

but in practice the serial and communication parts limit this behavior. As long as communication can be neglected, $f(n)$ approaches a plateau which is determined by T_s,

$$f(n) \leq 1 + \frac{T_m}{T_s} \quad . \tag{36}$$

Eventually, the communication time $T_c(n)$ starts to dominate and suppresses the achievable speedup again,

$$f(n) \to \frac{T_s + T_m}{T_c(n)} \to 0 \, . \tag{37}$$

Clearly, this behavior limits the number n of workers that we can efficiently employ for a given task.

The challenges of parallelization are thus twofold: (i) increase the fraction T_m/T_s by parallelizing all computing-intensive parts of the program. For instance, if T_s amounts to 0.1 % of T_m, the plateau is

reached for $n = 1000$ workers. (ii) make sure that at this saturation point, $T_c(n)$ is still negligible. This can be achieved by (a) choose a communication algorithm where T_c increases with a low power of n, or (b) reduce the prefactor in $T_c(n)$, which summarizes the absolute amount of communication and blocking per node.

We will later on in the benchmarking of our parallelization algorithm compare the speedup to Amdahl's law [46]. For this, we neglect the communication time in Eq. (34), and write the time executed by the parallelizable part as a fraction $p \cdot T$ of the total time $T = T_s + T_M$ of the serial process, while the non-parallelizable part is then $(1-p)T$. The speedup in Eq. (34) translates then into

$$f(n) = \frac{1}{(1-p) + \frac{p}{n}} \overset{n\to\infty}{\longrightarrow} \frac{1}{1-p} \quad . \tag{38}$$

In Sec. 4.5.5 we use Amdahl's law as a comparison for parallelizable parts of $p = 90\%$ and 100 %, respectively. Note that Amdahl's law is considered to be very critical on the possible speedup, while a more optimistic or realistic estimate is given by Gustafson's law [47]. To discuss the differences, however, is beyond the scope of this paper.

4.2 Computing tasks in WHIZARD

The computing tasks performed by WHIZARD vary, and crucially depend on the type and complexity of the involved physics processes. They also depend on the nature of the problem, such as whether it involves parton distributions or beam spectra, the generation of event files, or scans over certain parameters, or whether it is a LO or NLO process.

To begin with, we therefore identify the major parts of the program and break them down into sections which, in principle, can contribute to either T_s (serial), T_m (parallel), or T_c (communication).

SINDARIN. All user input is expressed in terms of SINDARIN expressions, usually collected in an input file. Interpreting the script involves pre-processing which partly could be done in parallel. However, the SINDARIN language structure allows for mixing declarations with calculation, so parallel pre-processing can introduce nontrivial communication. Since scripts are typically short anyway, we have not yet considered parallel evaluation in this area. This also applies for auxiliary calculations that are performed within SINDARIN expressions.

Models. Processing model definitions is done by programs external to the WHIZARD core in advance. We do not consider this as part of the WHIZARD workflow. Regarding reading and parsing the resulting model files by WHIZARD, the same considerations apply as for the SINDARIN input. Nevertheless, for complicated models such as the MSSM, the internal handling of model data involves lookup tables. In principle, there is room for parallel evaluation. This fact has not been exploited, so far, since it did not constitute a bottleneck.

Process construction. Process construction with WHIZARD, i.e., setting up data structures that enable matrix-element evaluation, is delegated to programs external to the WHIZARD core. For tree-level matrix elements, the in-house O'MEGA generator constructs Fortran code which is compiled and linked to the main program. For loop matrix elements, WHIZARD relies on programs such as GOSAM, RECOLA, or OPENLOOPS. The parallelization capabilities rely on those extra programs, and are currently absent. Therefore, process-construction time contributes to T_s only.

Phase-space construction. Up to WHIZARD version 2.6.1, phase-space construction is performed internally with WHIZARD (i.e. by the WHIZARD core), by a module which recursively constructs data structures derived from simplified Feynman graphs. The algorithm is recursive and does not lead to obvious parallelization methods; the resulting T_s contribution is one of the limiting factors.

A new algorithm, which is described below in Sec. 4.6, re-uses the data structures from process construction via O'MEGA. The current implementation is again serial (T_s), but significantly more efficient. Furthermore, since it does not involve recursion it can be parallelized if the need arises.

Integration. Integrating over phase space involves the VAMP algorithm as described above. In many applications, namely those with complicated multi-particle or NLO matrix elements, integration time dominates the total computing time. We can identify tasks that qualify for serial, parallel, and communication parts:

- Initialization. This part involves serial execution. If subsequent calculations are done in parallel, it also involves communication, once per process.
- Random-number generation. The VAMP integrator relies on random-number sequences. If we want parallel evaluation on separate workers, the random-number generator should produce independent, reproducible sequences without the necessity for communication or blocking.
- VAMP sampling. Separate sampling points involve independent calculation, thus this is a preferred target for turning serial into parallel evaluation. The (pseudo-) stratified algorithm involves some management, therefore communication may not be entirely avoidable.
- Phase-space kinematics. Multi-channel phase-space evaluation involves two steps: (i) computing the mapping from the unit hypercube to a momentum configuration, for a single selected phase-space channel, and (ii) computing the inverse mapping, for all other phase-space channels. The latter part is a candidate for parallel evaluation. The communication part involves distributing the momentum configuration for a single event. The same algorithm structure applies to the analogous discrete mappings introduced by the VAMP algorithm.
- Structure functions. The external PDF library for hadron collisions (LHAPDF) does not support intrinsic parallel evaluation. This also holds true for the in-house CIRCE1/CIRCE2 beamstrahlung library.
- Matrix-element evaluation. This involves sums over quantum numbers: helicity, color, and particle flavor. These sums may be distributed among workers. The tradeoff of parallel evaluation has to be weighted against the resulting communication. In particular, common subexpression elimination or caching partial results do optimize serial evaluation, but actually can inhibit parallel evaluation or introduce significant extra communication.
- Grid adaptation. For a grid adaptation step, results from all sampling points within a given iteration have to be collected, and the adapted grids have to be sent to the workers. Depending on how grids are distributed, this involves significant communication. The calculations for adapting grids consume serial time, which in principle could also be distributed.
- Integration results. Collecting those is essentially a byproduct of adaptation, and thus does not involve extra overhead.

Simulation. A simulation pass is similar to an integration pass. There is no grid adaptation involved. The other differences are

- Sampling is done in form of strict importance sampling. This is actually simpler than sampling for integration.
- Events are further transformed or analyzed. This involves simple kinematic manipulations, or complex calculations such as parton shower and hadronization. The modules that are used for such tasks, such as PYTHIA or WHIZARD's internal module, do not support intrinsic parallelization. Generating histograms and plots involves communication and some serial evaluation.
- Events are written to file. This involves communication and serial evaluation, either event by event, or by combining event files generated by distinct workers.

Rescanning events. In essence, this is equivalent to simulation. The difference is that the input is taken from an existing event file, which is scanned serially. If the event handling is to be distributed, there is additional communication effort.

Parameter scans. Evaluating the same process(es) for different sets of input data, can be done by scripting a loop outside of WHIZARD. In that case, communication time merely consists of distributing the input once, and collecting the output, e.g., fill plots or histograms. However, there are also contributions to T_s, such as compile time for process code. Alternatively, scans can be performed using SINDARIN loop constructs. Such loops may be run in parallel. This avoids some of the T_s overhead, but requires communicating WHIZARD internal data structures. Phase-space construction may contribute to either T_s or T_m, depending on which input differs between workers. Process construction and evaluation essentially

turns into T_m. This potential has not been raised yet, but may be in a future extension. The benefit would apply mainly for simple processes where the current parallel evaluation methods are not efficient due to a small T_m fraction.

4.3 Paradigms and tools for parallel evaluation

There are a number of well-established protocols for parallel evaluation. They differ in their overall strategy, level of language support, and hardware dependence. In the following, we list some widely used methods.

1. MPI (message-passing interface, cf. e.g. [48]). This protocol introduces a set of independent abstract workers which virtually do not share any resources. By default, the program runs on all workers simultaneously, with different data sets. (Newer versions of the protocol enable dynamic management of workers.) Data must be communicated explicitly via virtual buffers. With MPI 3, this communication can be set up asynchronously and non-blocking. The MPI protocol is well suited for computing clusters with many independent, but interconnected CPUs with local memory. On such hardware, communication time cannot be neglected.

   For `Fortran`, MPI is available in form of a library, combined with a special run manager.

2. OpenMP (open multi-processing, cf. e.g. [49]). This protocol assumes a common address space for data, which can be marked as local to workers if desired. There is no explicit communication. Instead, data-exchange constraints must be explicitly implemented in form of synchronization walls during execution. OpenMP thus maps the configuration of a shared-memory multi-core machine. We observe that with such hardware setup, communication time need not exceed ordinary memory lookup. On the other hand, parallel execution in a shared-memory (and shared-cache) environment can run into blocking issues.

   `Fortran` compilers support OpenMP natively, realized via standardized preprocessor directives.

3. Coarrays (cf. e.g. [50]). This is a recent native `Fortran` feature, introduced in the `Fortran2008` standard. The coarray feature combines semantics both from MPI and OpenMP, in the form that workers are independent both in execution and in data, but upon request data can be tagged as mutually addressable. Such addressing implicitly involves communication.

4. Multithreading. This is typically an operating-system feature which can be accessed by application programs. Distinct threads are independent, and communication has to be managed by the operating system and kernel.

The current strategy for parallel evaluation with WHIZARD involves MPI and OpenMP, either separately or in combination. We do not use coarrays, which is a new feature that did not get sufficient compiler support, yet [1]. On the other hand, operating system threads are rather unwieldy to manage, and largely superseded by the OpenMP or MPI protocols which provide abstract system-independent interfaces to this functionality.

4.4 Random numbers and parallelization

WHIZARD uses pseudo random numbers to generate events. Most random number generators have in common that they compute a reproducible stream of uniformly distributed random numbers $\{x_i\} \in (0, 1)$ from a given starting point (seed) and they have a relative large periodicity. In addition, the generated random numbers should not have any common structures or global correlations. To ensure these prerequisites different test suites exist based on statistical principles and other methods. One is the TESTU01 library implemented in `ANSI C` which contains several tests for empirical randomness [51]. A very extensive collection of tests is the DIE HARD suite [52], also known as DIE HARD 1, which contains e.g. the squeeze test, the overlapping sums test, the parking lot test, the craps test, and the runs test [53]. There is also a more modern version of this test suite, DIE HARDER or DIE HARD 2 [54] which contains e.g. the KNUTHRAN [55] and the RANLUX [56, 57] tests. Furthermore, the computation of the pseudo random numbers should add as less as possible computation time.

[1] There would of course be the possibility to have a special version of WHIZARD only available for the newest version(s) of compilers to test those features. This is part of a future project.

The default random number generator of WHIZARD is the TAO random number generator proposed by [55] and provided by VAMP. This generator passes the DIE HARD tests. It is based on a lagged Fibonacci sequence,

$$X_{n+1} = (X_{n-k} + X_{n-l}) \mod 2^{30}, \tag{39}$$

with lags $k = 100$ and $l = 37$ computing portable, 30-bit integer numbers. The computation needs a reservoir of random numbers of at least $k = 100$ which have to be prepared in advance. To ensure a higher computation efficiency, a reservoir of more than 1000 is needed. Furthermore, the TAO random numbers suffers from its integer arithmetic. In general on modern CPUs floating point arithmetic is faster and can be put in pipelines allowing terser computation.

In order to utilize the TAO generator for a parallelized application we have to either communicate each random number before or during sampling, both are expensive on time, or we have to prepare or, at least, guarantee independent streams of random numbers from different instances of TAO by initializing each sequence with different seeds. The latter is hardly feasible or even impossible to ensure for all combinations of seeds and number of workers. This and the (time-)restricted integer arithmetic render the TAO random number generator impractical for our parallelization task.

To secure independent (and still reproducible) random numbers during parallel sampling we have implemented the RNGstream algorithm by [58]. The underlying generator is a combined multiple-recursive generator, referred to as `MRG32k3a`, based on two multiple-recursive generators,

$$x_{1,n} = \left(1403580 x_{1,n-2} - 810728 x_{1,n-3}\right) \mod 4294967087, \tag{40}$$

$$x_{2,n} = \left(527612 x_{2,n-1} - 1370589 x_{2,n-3}\right) \mod 4294944443, \tag{41}$$

at the n-th step of the recursion with the initial seed $\vec{x}_{i,0} = (x_{i,-2}, x_{i,-1}, x_{i,0})^T, i \in \{1, 2\}$. The two states are then combined to produce a uniformly-distributed random number u_n as

$$z_n = \left(x_{1,n} - x_{2,n}\right) \mod 4292967087, \tag{42}$$

$$u_n = \begin{cases} z_n / 4294967088 & \text{if } z_n > 0. \\ 4292967087 / 4292967088 & \text{if } z_n = 0. \end{cases} \tag{43}$$

The resulting random number generator has a period of length $\approx 2^{191}$. It passes all tests of TESTU01 and DIE HARD.

The overall sequence of random numbers is divided into streams of length 2^{127}, each of these streams is then further subdivided into substreams of length 2^{76}. Each stream or subsequent substream can be accessed by repeated application of the transition function $x_n = T(x_{n-1})$. We rewrite the transition function as a matrix multiplication on a vector, making the linear dependence clear, $x_n = T \times x_{n-1}$. Using the power of modular arithmetic, the repated application of the transition function can be precomputed and be stored making access of the (sub-)streams as simple as sampling one. In the context of the parallel evaluation of the random number generator we can get either independent streams of random numbers for each worker, or, conserving the numerical properties for the integration process, assign each channel a stream and each stratification cell of the integration grid a substream in the serial and parallel run. Then we can easily distribute the workers among channels and cells without further concern about the random numbers.

The original implementation of the RNGstream was in `C++` using floating point arithmetic. We have rewritten the implementation for `Fortran2008` in WHIZARD.

4.5 Parallel evaluation in WHIZARD

To devise a strategy for parallel evaluation, we have analyzed the workflow and the scaling laws for different parts of the code, as described above. Complex multi-particle processes are the prime target of efficient evaluation. In general, such processes involve a large number of integration dimensions, a significant number of quantum-number configurations to be summed over, a large number of phase-space points per iteration of the integration procedure, and a large number of phase-space channels. By contrast, for a single phase-space channel the number of phase-space points remains moderate.

After the integration passes are completed, event generation in the simulation pass is another candidate for parallel execution. Again, a large number of phase-space points have to be sampled within the same computational model as during integration. Out of the generated sample of partonic events, in the unweighted mode, only a small fraction is further processed. The subsequent steps of parton shower, hadronization, decays, and file output come with their own issues of computing (in-)efficiency.

We address the potential for parallel evaluation by two independent protocols, OpenMP and MPI. Both frameworks may be switched on or off independent of each other.

4.5.1 Sampling with OpenMP

On a low-level scale, we have implemented OpenMP as a protocol for parallel evaluation. The OpenMP paradigm is intended to distribute workers among the physical core of a single computing node, where actual memory is shared between cores. While in principle, the number of workers can be set freely by the user of the code, one does expect improvements as long as the number of workers is less or equal to the number of physical cores. The number of OpenMP workers therefore is typically between 1 and 8 for standard hardware, and can be somewhat larger for specialized hardware.

We apply OpenMP parallelization for the purpose of running simple `Fortran` loops in parallel. These are

1. The loop over helicities in the tree-level matrix-element code that is generated by O'Mega. For a typical $2 \to 6$ fermion process, the number of helicity combinations is $2^8 = 64$ and thus fits the expected number of OpenMP workers. We do not parallelize the sum of the flavor or color quantum numbers. In the current model of O'Mega code, those sums are subject to common-subexpression elimination which inhibits trivial parallelization.

2. The loop over channels in the inverse mapping between phase-space parameters and momenta. Due to the large number of channels, the benefit is obvious, while the communication is minimal, and in any case is not a problem in a shared-memory setup.

3. Analogously, the loop over channels in the discrete inverse mapping of the phase-space parameters within the Vamp algorithm.

In fact, these loops cover the most computing-intensive tasks. As long as the number of OpenMP workers is limited, there is no substantial benefit from parallelizing larger portions of code at this stage.

4.5.2 Sampling with MPI

The MPI protocol is designed for computing clusters. We will give a short introduction into the terminology and the development of its different standards over time in the next subsection. The MPI model assumes that memory is local to each node, so data have to be explicitly transferred between nodes if sharing is required. The number of nodes can become very large. In practice, for a given problem, the degree of parallelization and the amount of communication limits the number of nodes where the approach is still practical. For Whizard, we apply the MPI protocol at a more coarse-grained level than OpenMP, namely the loop over sampling points which is controlled by the Vamp algorithm.

As discussed above, in general, for standard multi-particle problems the number of phase-space channels is rather large, typically exceeding $10^3 \ldots 10^4$. In that case, we assign one separate channel or one subset of channels to each worker. In some calculations, the matrix element is computing-intensive but the number of phase-space channels is small (e.g. NNLO virtual matrix elements), so this model does not apply. In that case, we parallelize over single grids. We assign to each worker a separate slice of the n_r^d cells of the stratification space. In principle, for the simplest case of $n_r = 2$, we can exploit up to 2^d computing nodes for a single grid. On the other hand, parallelization over the r-space is only meaningful *when* $n_r \geq 2$. Especially, when we take into account that n_r changes between different iterations as the number of calls N_C depends on the multi-channel weights α_i. Hence, we implement a sort of auto-balancing in the form that we choose between the two modes of parallelization before and during sampling in order to handle the different scenarios accordingly. Per default, we parallelize over phase-space multi-channel, but prefer single-grid parallelization for the case that the number of cells in r-space is $n_r > 1$. Because the single-grid parallelization is finer grained than the phase-space channel parallelization, this allows in principle to exploit more workers. Furthermore, we note that the Monte Carlo integration itself does not exhibit any boundary conditions demanding communication during sampling (except when we impose such a condition by hand). In particular, there is no need to communicate random numbers. We discuss the details of the implementation later on.

4.5.3 The Message-Passing Interface (MPI) Standard

We give a short introduction into the terminology necessary to describe our implementation below, and also into the message-passing interface (MPI) standard. The MPI standard specifies a large amount

of procedures, types, memory and process management and handlers, for different purposes. The wide range of functionality obscures a clear view on the problem of parallelization and on the other hand it unnecessarily complicates the problem itself. So, we limit the use of functionality to an absolute minimum. E.g., we do not make use of the MPI shared-memory model and, for the time being, the use of an own process management for a server-client model. In the following we introduce the most common terms. In the implementation details below, we again refer to the MPI processes as *workers* in order to not confuse them with the WHIZARD's physical processes.

The standard specifies MPI programs to consist of autonomous processes, each in principle running its own code, in an MIMD[2] style, cf. [59], p. 20. In order to abstract the underlying hardware and to allow separate communication between different program parts or libraries, the standard generalizes as *communicators* processes apart from the underlying hardware and introduces communication contexts. Context-based communication secures that different messages are always received and sent within their context, and do not interfere with messages in other contexts. Inside communicators, processes are grouped (process group) as ordered sets with assigned ranks (labels) $0, \ldots, n-1$. The predefined `MPI_COMM_WORLD` communicator contains all processes known at the initialization of a MPI program in a linearly ordered fashion. In most cases, the linear order does not reflect the architecture of the underlying hardware and network infrastructure, therefore, the standard defines the possibility to map the processes onto the hardware and network infrastructure to optimize the usage of resources and increase the speedup.

A way to conceivably optimize the parallelization via MPI is to make the MPI framework aware about the communication flows in your application. In the group of processes in a communicator, not all processes will communicate with every other process. The network of inter-process communication is known as MPI *topologies*. The default is `MPI_UNDEFINED` where not specific topology has been specified, while `MPI_CART` is a Cartesian (nearest-neighbor) topology. Special topologies can be defined with `MPI_GRAPH`. In this paper we only focus on the MPI parallelization of the Monte Carlo VAMP. A specific profiling of run times of our MPI parallelization could reveal specific topological structures in the communication which might offer potential for improvement of speedups. This, however, is beyond the scope of this paper.

Messages are passed between sender(s) and receiver(s) inside a communicator or between communicators where the following communication modes are available:

non-blocking A non-blocking procedure returns directly after initiating a communication process. The status of communication must be checked by the user.

blocking A blocking procedure returns after the communication process has completed.

point-to-point A point-to-point procedure communicates beetween a single receiver and single sender.

collective A collective procedure communicates with the complete process group. Collective procedures must appear in the same order on all processes.

The standard distinguishes between blocking and non-blocking point-to-point or collective communications. A conciser program flow and an increased speedup are advantages of non-blocking over blocking communication.

In order to ease the startup of a parallel application, the standard specifies the startup command `mpiexec`. However, we recommend the de-facto standard startup command `mpirun` which is in general part of a MPI-library. In this way, the user does not have to bother with the quirks of the overall process management and inter-operability with the operating system, as this is then covered by `mpirun`. Furthermore, most MPI-libraries support interfaces to cluster software, e.g. SLURM, TORQUE, HTCONDOR.

In summary, we do not use process and (shared-)memory management, topologies and the advanced error handling of MPI, which we postpone to a future work.

4.5.4 Implementation Details of the MPI Parallelization

In this subsection, we give a short overview of the technical details of the implementation to show and explain how the algorithm works in detail.

In order to minimize the communication time T_c, we only communicate runtime data which cannot be provided *a priori* by WHIZARD's pre-integration setup through the interfaces of the integrators. Furthermore, we expect that the workers are running identical code up to different communication-based code branches. The overall worker setup is externally coordinated by the MPI-library provided process

[2] Multiple instructions, multiple data. Machines supporting MIMD have a number of processes running asynchronously and independently.

manager `mpirun`. As mentioned in the last subsection, most MPI libraries have interfaces supporting cluster software to automatically do the steering between the different nodes, e.g. SLURM, TORQUE or HTCONDOR.

In order to enable file input/output (I/O), in particular to allow the setup of a process, without user intervention, we implement the well-known master-slave model. The master worker, specified by rank 0, is allowed to setup the matrix-element library and to provide the phase-space configuration (or to write the grid files of VAMP) as those are involved with heavy I/O operations. The other workers function solely as slave workers supporting only integration and event generation. Therefore, the slave workers have to wait during the setup phase of the master worker. We implement this dependence via a blocking call to `MPI_BCAST` for all slaves while the master is going through the setup steps. As soon as the master worker has finished the setup, the master starts to broadcast a simple logical which completes the blocked communication call of the slaves allowing the execution of the program to proceed. The slaves are then allowed to load the matrix-element library and read the phase-space configuration file *in parallel*. The slave setup adds a major contribution to the serial time, mainly out of our control as the limitation of the parallel setup of the slave workers are imposed by the underlying filesystem and/or operating system, since all the workers try to read the files simultaneously. We expect that the serial time is increased at least by the configuration time of WHIZARD without building and making the matrix-element library and configuring the phase-space. Therefore, we expect the configuration time at least to increase linearly with the number of workers.

In the following, we outline the reasoning and implementation details. At the beginning of an iteration pass of VAMP, we broadcast the current grid setup and the channel weights from the master to all slave workers. For this purpose, the MPI protocol defines collective procedures, e.g. `MPI_BCAST` for broadcasting data from one process to all other processes inside the `communicator`. The multi-channel formulas Eq. (28) and Eq. (30) force us to communicate each grid[3] to every worker. The details of an efficient communication algorithm and its implementation is part of the actual MPI implementation (most notably the OPENMPI and MPICH libraries) and no concern of us. After we have communicated the grid setup using the collective procedure `MPI_BCAST`, we let each worker sample over a predefined set of phase-space channels. Each worker skips its non-assigned channels and advances the stream of random numbers to the next substream such as it would have used them for sampling. However, if we can defer the parallelization to VEGAS, we spawn a `MPI_BARRIER` waiting for all other workers to finish their computation until the call of the barrier and start with parallelization of the channel over its grid. When all channels have been sampled, we collect the results from every channel and combine them to the overall estimate and variance. We apply a master/slave chain of communication where each slave sends his results to the master as shown in Listing 1. For this purpose the master worker and the slave worker execute different parts of the code. The computation of the final results of the current pass is then exclusively done by the master worker. Additionally, the master writes the results to a VAMP grid file for the case that the computation is interrupted and should be restarted after the latest iteration (adding extra serial time to WHIZARD runs).

```
subroutine vamp2_integrate_collect ()
  ! ...
  do ch = 1, self%config%n_channel
     if (self%integrator(ch)%is_parallelizable ()) cycle
     worker = map_channel_to_worker (ch, n_size)
     result = self%integrator(ch)%get_result ()
     if (rank == 0) then
        if (worker /= 0) then
           call result%receive (worker, ch)
           call self%integrator(ch)%receive_distribution (worker, ch)
           call self%integrator(ch)%set_result (result)
        end if
     else
        if (rank == worker) then
           call result%send (0, ch)
           call self%integrator(ch)%send_distribution (0, ch)
        end if
     end if
  end do
  ! ...
end subroutine vamp2_integrate_collect
```

Listing 1: *Collecting the results of the multi-channel computation on the master worker with rank* 0.

[3]The grid type holds information on the binning x_i, the number of dimensions, the integration boundaries and the jacobian.

In order to employ the VEGAS parallelization from [60] we divide the r space into a parallel subspace $r_\parallel$ with dimension $d_\parallel = \lfloor d/2 \rfloor$ over which we distribute the workers. We define the left-over space $r_\perp = r \setminus r_\parallel$ as perpendicular space with dimension $d_\perp = \lceil d/2 \rceil$. Assigning to each worker a subspace $r_{\parallel,i} \subset r_\parallel$, the worker samples $r_{\parallel,i} \otimes r_\perp$. For the implementation we split the loop over the cells in r-space into an outer parallel loop and an inner perpendicular loop. In the outer parallel loop the implementation descends in the inner loop only when worker and corresponding subspace match, if not, we advance the state of the

```
  if (self%is_parallelizable ()) then
     do k = 1, rank
        call increment_box_coord (self%box(1:n_dim_par), box_success)
        if (.not. box_success) exit
     end do
     select type (rng)
     type is (rng_stream_t)
        call rng%advance_state (self%config%n_dim * self%config%calls_per_box&
             & * self%config%n_boxes**(self%config%n_dim - n_dim_par) * rank)
     end select
  end if
  loop_over_par_boxes: do while (box_success)
     loop_over_perp_boxes: do while (box_success)
        do k = 1, self%config%calls_per_box
           call self%random_point (rng, x, bin_volume)
           fval = self%jacobian * bin_volume * func%evaluate (x)
           ! ...
        end do
        ! ...
        call increment_box_coord (self%box(n_dim_par + 1:self%config&
             &%n_dim), box_success)
     end do loop_over_perp_boxes
     shift: do k = 1, n_size
        call increment_box_coord (self%box(1:n_dim_par), box_success)
        if (.not. box_success) exit shift
     end do shift
     if (self%is_parallelizable ()) then
        select type (rng)
        type is (rng_stream_t)
           call rng%advance_state (self%config%n_dim * self%config%calls_per_box&
                & * self%config%n_boxes**(self%config%n_dim - n_dim_par) * (n_size - 1))
        end select
     end if
  end do loop_over_par_boxes
```

Listing 2: *Iteration over r-space and advancing the random number generator.*

random number generator by the number of sample points in $r_{\parallel,i} \otimes r_\perp$, where i is the skipped outer loop index.

After sampling over the complete r-space the results of the subsets are collected. All results are collected by reducing them by an operator, e.g. `MPI_SUM` or `MPI_MAX` with `MPI_REDUCE` (reduction here is meant as a concept from functional programming where data reduction is done by reducing a set of numbers into a smaller set of numbers via a function or an operator). The application of such a procedure from a MPI library is in general more efficient than a self-written procedure. We implemented all communication calls as non-blocking, i.e. the called procedure will directly return after setting up the communication. The communication itself is done in the background, e.g. by an additional communication thread. The details are provided in the applied MPI library. To ensure the completion of communication a call to `MPI_WAIT` has to be done.

When possible, we let objects directly communicate by `Fortran2008` type-bound procedures, e.g. the main VEGAS grid object, `vegas_grid_t` has `vegas_grid_broadcast`. The latter broadcasts all relevant grid information which is not provided by the API of the integrator. We have to send the number of bins to all processes before the actual grid binning happnes, as the size of the grid array is larger than the actual number of bins requested by VEGAS.[4]

```
subroutine vegas_grid_broadcast (self)
  class(vegas_grid_t), intent(inout) :: self
  integer :: j, ierror
  type(MPI_Request), dimension(self%n_dim + 4) :: status
```

[4]The size of the grid array is set to a pre-defined or user-defined value. If only the implementation switches to stratified sampling, the number of bins is adjusted to the number of boxes/cells and, hence, does not necessarily match the size of the grid array.

```
  ! Blocking
  call MPI_Bcast (self%n_bins, 1, MPI_INTEGER, 0, MPI_COMM_WORLD)
  ! Non blocking
  call MPI_Ibcast (self%n_dim, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, status(1))
  call MPI_Ibcast (self%x_lower, self%n_dim, &
       & MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, status(2))
  call MPI_Ibcast (self%x_upper, self%n_dim, &
       & MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, status(3))
  call MPI_Ibcast (self%delta_x, self%n_dim, &
       & MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, status(4))
  ndim: do j = 1, self%n_dim
     call MPI_Ibcast (self%xi(1:self%n_bins + 1, j), self%n_bins + 1,&
          & MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, status(4 + j))
  end do ndim
  call MPI_Waitall (self%n_dim + 4, status, MPI_STATUSES_IGNORE)
end subroutine vegas_grid_broadcast
```

Listing 3: *Broadcast grid information using blocking and non-blocking procedures.*

Further important explicit implementations are the type-bound procedures `vegas_send_distribution`/`vegas_receive_distribution` and `vegas_result_send`/`vegas_result_receive` which are needed for the communication steps involved in VAMP in order to keep the VEGAS integrator objects encapsulated (i.e. preserve their `private` attribute).

Beyond the inclusion of non-blocking collective communication we choose as a minimum prerequisite the major version 3 of MPI for better interoperability with `Fortran` and its conformity to the `Fortran2008+TS19113` (and later) standard [59], Sec. 17.1.6. This, e.g., allows for MPI-derived type comparison as well as asynchronous support (for I/O or non-blocking communication).

A final note on the motivation for the usage of non-blocking procedures. Classic (i.e. serial) Monte Carlo integration exhibits no need for in-sampling communication in contrast to classic application of parallelization, e.g. solving partial differential equations. For the time being, we still use non-blocking procedures in VEGAS for future optimization, but in a more or less blocking fashion, as most non-blocking procedures are followed by a `MPI_WAIT` procedure. However, the multi-channel ansatz adds sufficient complexity, as each channel itself is an independent Monte Carlo integration. A typical use case is the collecting of already sampled channels while still sampling the remaining channels as it involves the largest data transfers in the parallelization setup. Here, we could benefit most from non-blocking communication. To implement these procedures as non-blocking necessitates a further refactoring of the present multi-channel integration of WHIZARD, because in that case the master worker must not perform any kind of calculation but should only coordinate communication. A further constraint to demonstrate the impact of turning many of our still blocking communication into a non-blocking one is the fact that at the current moment, there do not exist any profilers compliant with the MPI3.1 status that support `Fortran2008`. Therefore, we have to postpone the opportunity to show the possibility of completely non-blocking communication in our setup.

4.5.5 Speedup and Results

In order to assess the efficiency of our parallelization, we compare the two modes, the traditional serial VAMP implementation and our new parallelized implementation. In the following, we study different processes at different levels of complexity in order to investigate the scaling behavior of our parallel integration algorithm. The process $e^+e^- \to \mu^+\mu^-$ at energies below the Z resonance has only one phase-space channel (s-channel photon exchange and its integration is adapted over one grid. Parallelization is done in VEGAS over stratification space. The process $e^+e^- \to \mu^+\mu^-\mu^+\mu^-\nu_\mu\bar{\nu}_\mu$ with its complicated vector-boson interactions gives rise to $\mathcal{O}(3000)$ phase-space channels. With overall $\mathcal{O}(10^6)$ number of calls for the process, each phase-space channel is sampled (in average) by $\mathcal{O}(10^2)$ calls suppressing the stratification space of all grids. Therefore, parallelization is done over the more coarse phase-space channel loop. The last process we investigate, $e^+e^- \to \mu^+\mu^-\mu^+\mu^-$, has $\mathcal{O}(100)$ phase-space channels where we expect for some grids a distinct stratification space (at least two cells per dimension) allowing WHIZARD to switch between VEGAS and multi-channel parallelization. All but the first trivial examples are taken from [3] mimicking real-world application. The results are shown in Fig. 1.

Furthermore, we are interested in the behavior for increasing complexity of a single process, e.g. increasing (light) flavor content of processes with multiple jets. For the two processes, $jj \to W^-(\to e^-\bar{\nu}_e) + \{j, jj\}$ we increase the number of massless quark flavors in the content of the jets. The results in Fig. 2 indicate that for a single final-state jet more flavor content (and hence more complicated matrix elements) lead to lower speedups. For two (and more) final state jets the speedups increases with the

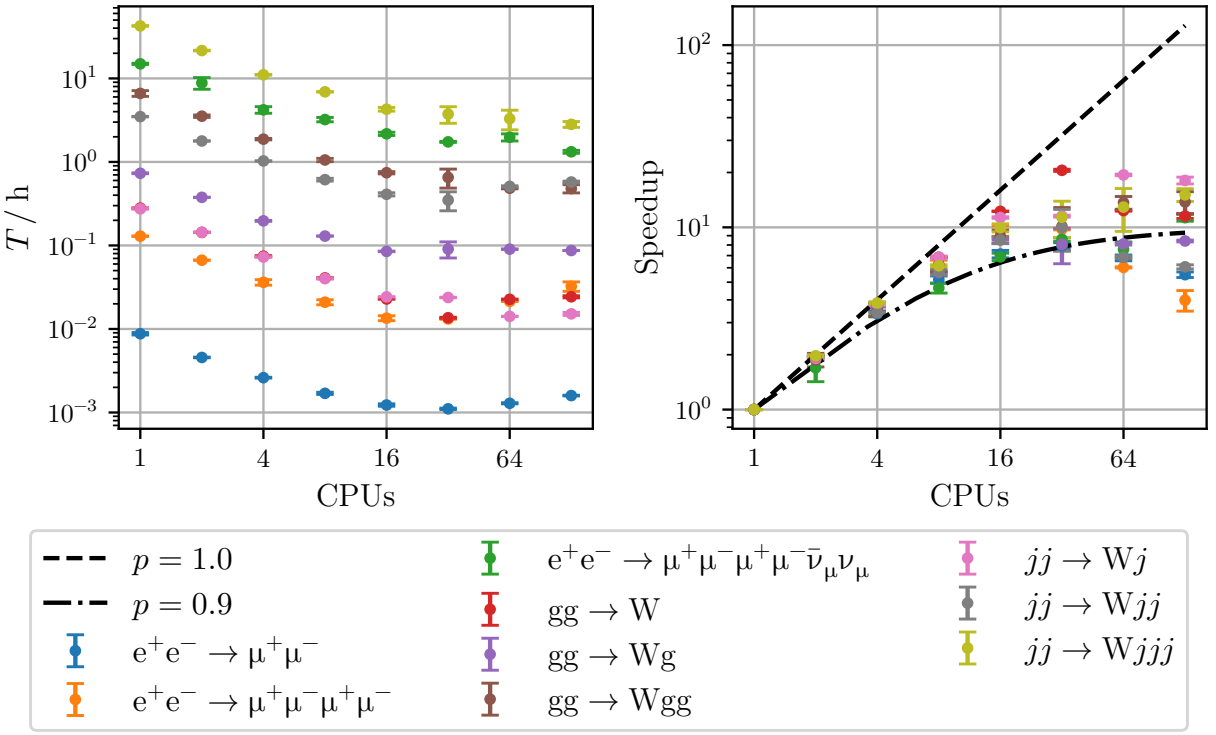

Figure 1: *We show the overall computation time for different complex processes for different numbers of participating CPUs (left panel). The numbers of CPUs is chosen as a power of 2. In the right panel, we plot the speedup of the processes and compare them to ideal of Amdahl's Law with parallelizable fractions of 1.0 (dashed) and 0.9 (dash-dotted).*

multiplicity of light quarks in the jet definition. This means that possibly for smaller matrix elements there is a communication overhead when increasing the complexity of the matrix element, while for the higher multiplicity process and many more phase-space channels, improvement in speedup can be achieved.

We benchmark the processes on the high performance cluster of the University of Siegen (Hochleistungrechner Universität Siegen, HorUS) which provides 34 Dell PowerEdge C6100 each containing 4 computing nodes with 2 CPUs. The nodes are connected by gigabit ethernet and the CPUs are Intel Xeon X5650 with 6 cores each 2.7 GHz and 128 MiB cache. We employ two different WHIZARD builds, a first one only with OPENMPI 2.1.2, and a second one with additional OpenMP support testing the hybrid parallelization. The HorUS cluster utilizes SLURM 17.02.2 as batch and allocation system allowing for easy source distribution. We run WHIZARD using the MPI-provided run manager `mpirun` and measure the run-time with the GNU time command tool, and we average over three independent runs for the final result. We measure the overall computation time of a WHIZARD run including the complete process setup with matrix-element generation and phase-space configuration. It is expected that the setup step gives rise to the major part of the serial computation of WHIZARD, and also the I/O operations of the multi-channel integrator, which saves the grids after each integration iteration. As this is a quasi-standard, we benchmark over N_{CPU} in powers of 2. Given the architecture of the HorUS cluster with its double hex-cores, benchmarking in powers of 6 would maybe be more appropriate for the MPI-only measurements. We apply a node-specific setup for the measurement of the hybrid parallelization. Each CPU can handle up to six threads without any substantial throttling. We operate over $N_{\text{Threads}} = \{1, 2, 3, 6\}$ with either fixed overall number of involved cores, $N_{\text{Worker}} = \{60, 30, 20, 10\}$, with results shown in Fig. 3, or with fixed number of workers $N_{\text{Worker}} = 20$, with results shown in Fig. 4.

Coming back to Fig. 1 showing the results of the benchmark measurement for MPI: The process $e^+e^- \to \mu^+\mu^-$ saturates for $N > 32$. The serial runtime of WHIZARD is dominating for that process with its two-dimensional integration measure (without beam structure functions) where the MC integration is anyways inferior to classical integration techniques. The process $e^+e^- \to \mu^+\mu^-\mu^+\mu^-$ showing mixed multi-channel and VEGAS parallelization, however, also saturates for $N > 32$. The multi-channel parallelizable process $e^+e^- \to \mu^+\mu^-\mu^+\mu^-\nu_\mu\bar{\nu}_\mu$ achieves a higher speedup but with decreasing slope. The overall speedup plot indicates a saturation beginning roughly at $N > 32$ where serial time and communication start

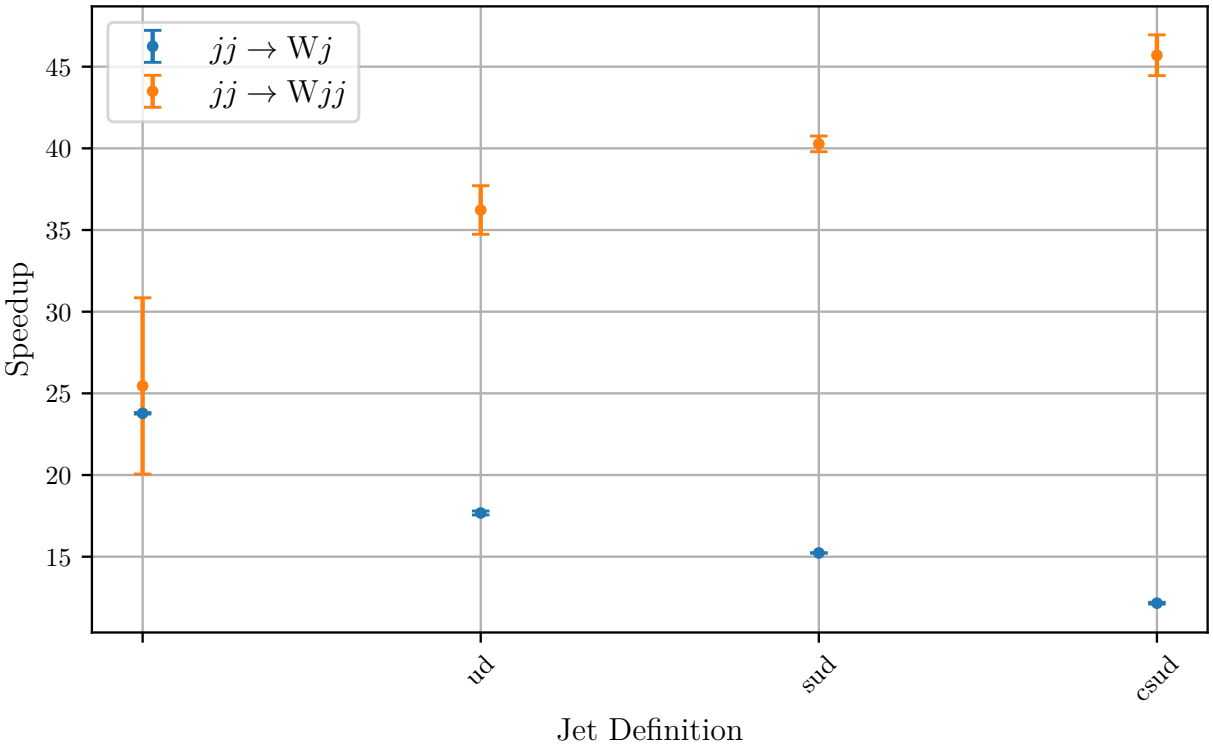

Figure 2: *We show the speedup for a jet process with increasing flavor content and a fixed numbers of CPUs of 60.*

to dominate. We conclude that WHIZARD embarks a parallelization fraction higher than at least 90 % for MPI. In the appendix A we present tables that show the actual physical runtimes for the different processes under consideration.

4.6 Alternative algorithm for phase-space generation

Profiling of the code reveals that for the moment the main bottleneck that inhibits speedups beyond $n = 100$ is the initial construction of the phase-space configurations, i.e. the phase-space channels and their parameterizations (the determination of the best mappings to be done for each channel or class of channels) which WHIZARD constructs from forests of Feynman tree-graphs. Resembling the language of Ref. [61], this construction algorithm is called *wood*. This *wood* algorithm takes into account the model structure, namely the three-point vertices to find resonant propagators, the actual mass values (to find collinear and soft singularities and to map mass edges), and the process energy. It turns out that while the default algorithm used in WHIZARD yields a good sample of phase-space channels to enable adaptive optimization, it has originally not been programmed in an efficient way. Though it is a recursive algorithm, it does not work on directed acyclical graphs (DAGs) like O'MEGA to avoid all possible redundancies.

Therefore, a new algorithm, *wood2*, has been designed in order to overcome this problem. Instead of constructing the forest of parameterizations from the model, it makes use of the fact that the matrix elements constructed optimally by O'MEGA in form of a DAG already contain all the necessary information with the exception of the numerical values for masses and collider energy. Thus, instead of building up the forest again, the algorithm takes a suitable description of the set of trees from O'MEGA and applies the elimination and simplification algorithm, in order to yield only the most relevant trees as phase-space channels. As it turns out, even in a purely serial mode, the new implementation performs much better and thus eliminates the most important source of saturation in speedup. Another benefit of the new algorithm is that it is much less memory-hungry than the original one which could have become a bottleneck for the traditional algorithm for complicated processes in very complicated models (e.g. extensions of the MSSM).

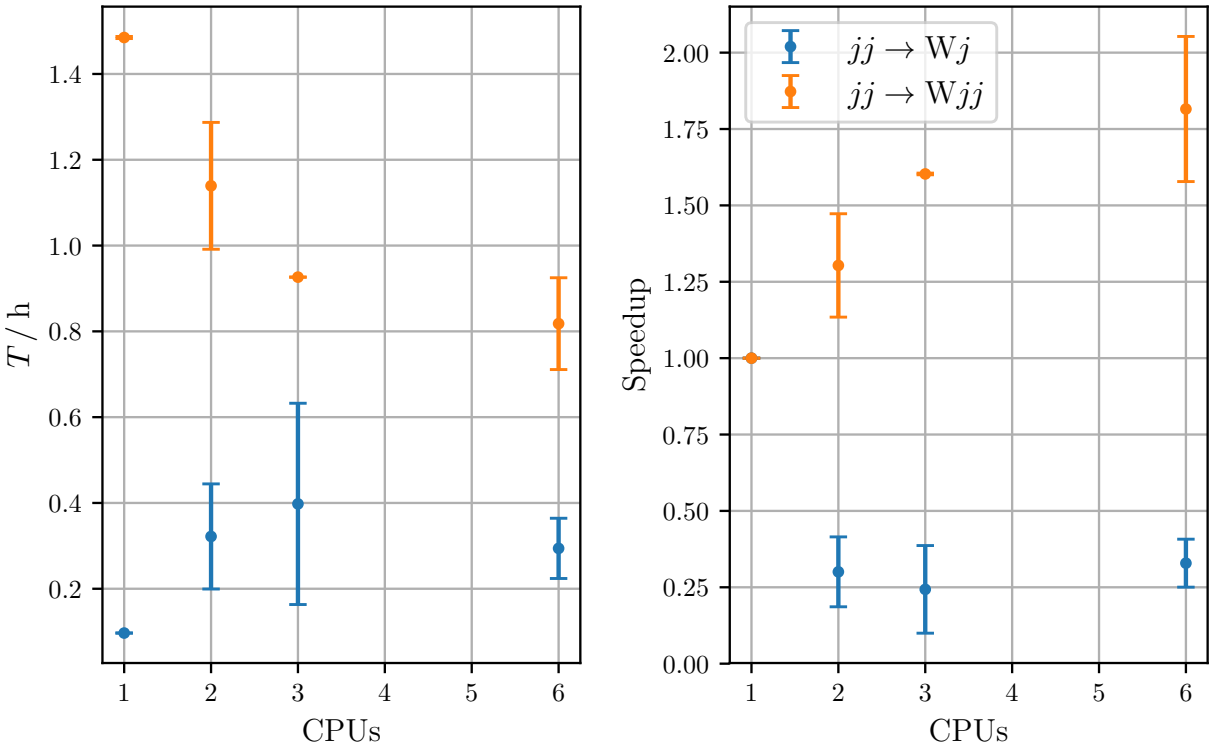

Figure 3: *These panels show the speedup for a fixed number of workers with increasing number of threads.*

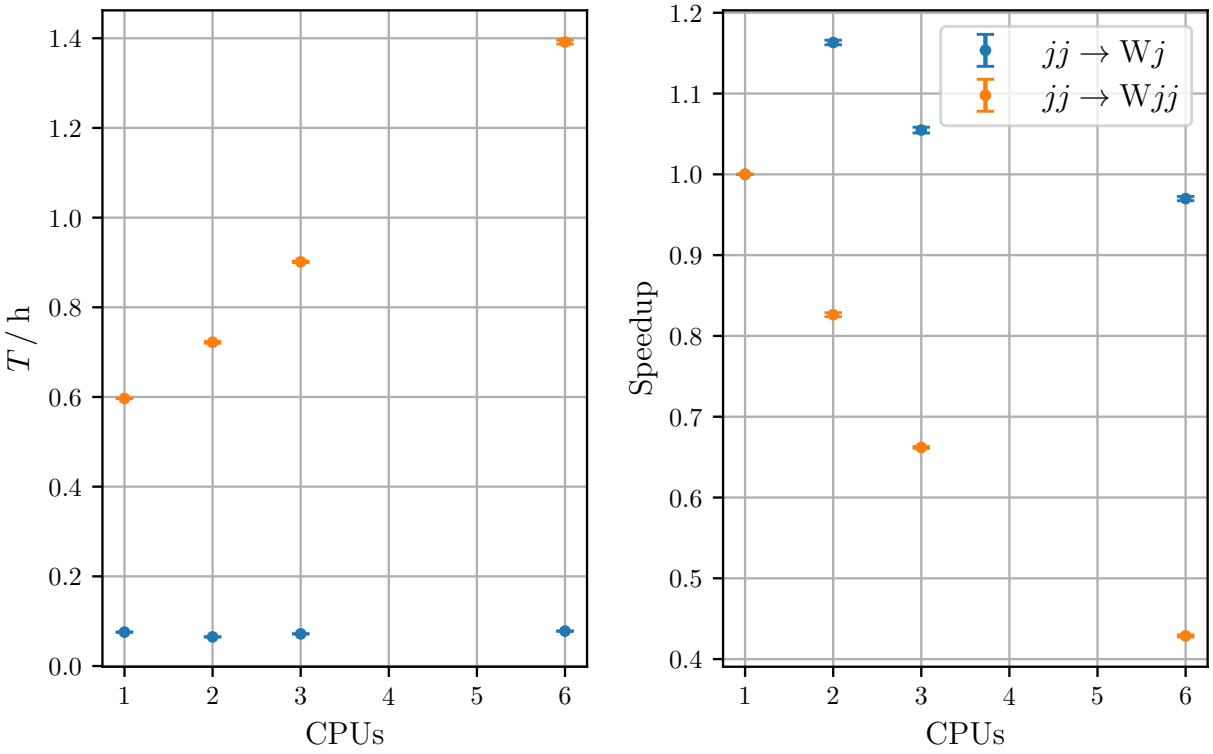

Figure 4: *We show the speedup at an overall fixed numbers of CPUs (60) involved in the parallelization. We distribute the CPUs among MPI and OpenMP parallelization. For the latter we have to respect the node structure where each node consists of two CPUs each with 6 cores handling up to 6 threads without any performance penalty.*

5 Conclusions and Outlook

Monte-Carlo simulations of elementary processes are an essential prerequisite for successful physics analyses at present and future colliders. High-multiplicity processes and high precision in signal and background detection put increasing demands on the required computing resources. One particular bottleneck is the multi-dimensional phase-space integration and the task of automatically determining a most efficient sampling for (unweighted) event generation. In this paper, we have described an efficient algorithm that employs automatic iterative adaptation in conjunction with parallel evaluation of the numerical integration and event generation.

The parallel evaluation is based on the paradigm of the Message Passing Interface Protocol (MPI) in conjunction with OpenMP multi-threading. For the concrete realization, the algorithm has been implemented within the framework of the multi-purpose event generator WHIZARD. The parallelization support for MPI or OpenMP can be selected during the configure step of WHIZARD. The new code constitutes a replacement module for the VAMP adaptive multi-channel integrator which makes active use of modern features in the current MPI-3.1 standard. Our initial tests for a variety of benchmark physics processes demonstrate a speedup by a factor > 10 with respect to serial evaluation. The best results have been achieved by MPI parallelization. The new implementation has been incorporated in the release version WHIZARD v2.6.4.

We were able to show that, in general, hybrid parallelization with OPENMP and MPI leads to a speedup which is comparable to MPI parallelization alone. However, combining both approaches is beneficial for tackling memory-intense processes, such as 8- or 10-particle processes. Depending on a particular computing-cluster topology, the latter approach can allow for a more efficient use of the memory locally available at a computing node. In the hybrid approach, WHIZARD is parallelized on individual multi-core nodes via OPENMP multithreading, while distinct computing nodes communicate with each other via MPI. The setup of the system allows for sufficient flexibility to make optimal use of both approaches for a specific problem.

The initial tests point to further possibilities for improvement, which we foresee for future development and refinements of the implementation. A server/client structure should give the freedom to re-allocate and assign workers dynamically during a computing task, and thus make a more efficient use of the available resources. Further speedup can be expected from removing various remaining blocking communications and replacing them by non-blocking communication while preserving the integrity of the calculation. Finally, we note that the algorithm shows its potential for calculations that spend a lot of time in matrix-element evaluation. For instance, in some tests of NLO QCD processes we found that the time required for integration could be reduced from the order of a week down to a few hours. We defer a detailed benchmarking of such NLO processes to a future publication.

6 Acknowledgements

The authors want to thank Bijan Chokoufé Nejad, Stefan Hoeche and Thorsten Ohl for helpful and interesting discussions. For their contributions to the new phase-space construction algorithm (known as *wood2*) we give special credits to Manuel Utsch and Thorsten Ohl.

A Results

The measured duration of the benchmarks are listed in the tables 2, 3, 4 and 5. It can be seen that the processes with a longer duration are subject to a large run-time fluctuation. We were able to reproduce this behavior with a variety of processes and attribute it back to the messy benchmark environment. The HoRUS of the University of Siegen is a university-wide cluster and is usually well utilized by other research groups, too. Therefore, a clean and reproducible measurement of longer runtime was difficult to make because on average we do not have the necessary computing power or network performance for us alone, which is reflected in particular in longer runtime.

N_{CPU}	$T(\text{e}^-\text{e}^+ \to \mu^-\mu^+)\text{s}^{-1}$	$T(\text{e}^-\text{e}^+ \to \mu^-\mu^+\mu^-\mu^+)\text{s}^{-1}$	$T(\text{e}^-\text{e}^+ \to \mu^-\mu^+\mu^-\mu^+\nu_\mu\overline{\nu}_\mu)\text{s}^{-1}$
1	31.6 ± 1.1	465.0 ± 2.9	$53\,813.8 \pm 920.4$
2	16.4 ± 0.1	239.8 ± 0.4	$31\,768.7 \pm 5068.0$
4	9.4 ± 0.1	130.3 ± 9.9	$15\,145.2 \pm 1369.9$
8	6.1 ± 0.2	75.3 ± 5.2	$11\,579.7 \pm 695.5$
16	4.4 ± 0.1	48.5 ± 3.3	7815.4 ± 346.6
32	4.0 ± 0.1	47.5 ± 0.4	6257.2 ± 87.4
64	4.6 ± 0.1	77.1 ± 0.3	7110.4 ± 693.1
128	5.7 ± 0.1	116.7 ± 15.1	4763.8 ± 190.7

Table 2: *Computation time of the leptonic processes.*

N_{CPU}	$T(jj \to \text{W}j)\text{s}^{-1}$	$T(jj \to \text{W}jj)\text{s}^{-1}$	$T(jj \to \text{W}jjj)\text{s}^{-1}$
1	987.8 ± 9.4	$12\,561.4 \pm 77.6$	$153\,072.1 \pm 1498.4$
2	517.2 ± 4.7	6413.5 ± 31.2	$77\,640.6 \pm 686.9$
4	261.5 ± 1.2	3699.7 ± 1.5	$39\,812.5 \pm 389.8$
8	144.2 ± 1.9	2208.6 ± 83.4	$24\,867.7 \pm 196.0$
16	87.3 ± 0.1	1475.3 ± 65.0	$15\,390.6 \pm 769.6$
32	85.7 ± 0.7	1258.0 ± 321.8	$13\,475.7 \pm 3025.9$
64	50.9 ± 0.1	1834.3 ± 30.4	$11\,854.4 \pm 3133.5$
128	54.6 ± 2.3	2069.1 ± 51.9	$10\,152.1 \pm 813.5$

Table 3: *Computation time of* $jj \to W^-(\to e^-\bar{\nu}_e) + nj$ *processes.*

N_{CPU}	$T(\text{gg} \to \text{W}\,q\bar{q})\text{s}^{-1}$	$T(\text{gg} \to \text{W}\,q\bar{q}\text{g})\text{s}^{-1}$	$T(\text{gg} \to \text{W}\,q\bar{q}\text{gg})\text{s}^{-1}$
1	1008.0 ± 1.6	2634.2 ± 30.0	$23\,823.2 \pm 1876.5$
2	516.4 ± 1.4	1357.4 ± 0.5	$12\,712.5 \pm 425.1$
4	267.8 ± 3.9	709.1 ± 9.6	6745.1 ± 137.5
8	147.0 ± 0.8	466.1 ± 1.0	3800.5 ± 174.3
16	82.3 ± 0.3	305.8 ± 2.5	2673.5 ± 83.5
32	49.0 ± 0.7	326.0 ± 71.2	2352.7 ± 603.0
64	81.6 ± 0.6	324.9 ± 2.6	1743.3 ± 27.8
128	87.5 ± 2.5	313.0 ± 0.3	1727.5 ± 194.9

Table 4: *Computation time of* $gg \to W^-(\to e^-\bar{\nu}_e)q\bar{q} + ng$ *processes.*

process	N_{CPU}	$T(\{\text{d}\})\text{s}^{-1}$	$T(\{\text{u},\text{d}\})\text{s}^{-1}$	$T(\{\text{u},\text{d},\text{s}\})\text{s}^{-1}$	$T(\{\text{u},\text{d},\text{s},\text{c}\})\text{s}^{-1}$
$jj \to \text{W}j$	1	1589.1±3.0	2189.3±8.3	2730.9±1.2	3440.6±11.2
	60	66.8±0.0	123.8±0.7	179.3±0.1	283.1±0.8
$jj \to \text{W}jj$	1	11 075.0±2347.7	29 405.0±1206.9	49 151.5±585.6	91 677.3±2502.7
	60	435.1±0.3	811.7±0.4	1220.4±0.8	2006.0±0.5

Table 5: *Computation time over increasing flavor content. The upper two lines are for the process* $pp \to Wj$*, the lower two for the process* $pp \to Wjj$*, respectively. The second column gives the number of CPU cores, the following columns are the results for an increasing number of massless quark flavors in the initial state and jets, growing from one* (d) *to four* (d, u, s, c)*.*

References

[1] T. Gleisberg et al. "Event generation with SHERPA 1.1". In: *JHEP* 02 (2009), p. 007. DOI: 10.1088/1126-6708/2009/02/007. arXiv: 0811.4622 [hep-ph].

[2] J. Alwall et al. "The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations". In: *JHEP* 07 (2014), p. 079. DOI: 10.1007/JHEP07(2014)079. arXiv: 1405.0301 [hep-ph].

[3] Wolfgang Kilian, Thorsten Ohl, and Jürgen Reuter. "WHIZARD: Simulating Multi-Particle Processes at LHC and ILC". In: *Eur. Phys. J.* C71 (2011), p. 1742. DOI: 10.1140/epjc/s10052-011-1742-y. arXiv: 0708.4233 [hep-ph].

[4] Neil D. Christensen et al. "Introducing an interface between WHIZARD and FeynRules". In: *Eur. Phys. J.* C72 (2012), p. 1990. DOI: 10.1140/epjc/s10052-012-1990-5. arXiv: 1010.3251 [hep-ph].

[5] Mauro Moretti, Thorsten Ohl, and Jurgen Reuter. "O'Mega: An Optimizing matrix element generator". In: (2001), pp. 1981–2009. arXiv: hep-ph/0102195 [hep-ph].

[6] Thorsten Ohl and Jurgen Reuter. "Clockwork SUSY: Supersymmetric Ward and Slavnov-Taylor identities at work in Green's functions and scattering amplitudes". In: *Eur. Phys. J.* C30 (2003), pp. 525–536. DOI: 10.1140/epjc/s2003-01301-7. arXiv: hep-th/0212224 [hep-th].

[7] Thorsten Ohl and Jurgen Reuter. "Testing the noncommutative standard model at a future photon collider". In: *Phys. Rev.* D70 (2004), p. 076007. DOI: 10.1103/PhysRevD.70.076007. arXiv: hep-ph/0406098 [hep-ph].

[8] Kaoru Hagiwara et al. "Supersymmetry simulations with off-shell effects for CERN LHC and ILC". In: *Phys. Rev.* D73 (2006), p. 055005. DOI: 10.1103/PhysRevD.73.055005. arXiv: hep-ph/0512260 [hep-ph].

[9] W. Kilian et al. "QCD in the Color-Flow Representation". In: *JHEP* 10 (2012), p. 022. DOI: 10.1007/JHEP10(2012)022. arXiv: 1206.3700 [hep-ph].

[10] Bijan Chokoufe Nejad, Thorsten Ohl, and Jürgen Reuter. "Simple, parallel virtual machines for extreme computations". In: *Computer Physics Communications* 196 (Nov. 2015), pp. 58–69. DOI: 10.1016/j.cpc.2015.05.015. URL: https://doi.org/10.1016%2Fj.cpc.2015.05.015.

[11] Wolfgang Kilian et al. "An Analytic Initial-State Parton Shower". In: *JHEP* 04 (2012), p. 013. DOI: 10.1007/JHEP04(2012)013. arXiv: 1112.1039 [hep-ph].

[12] Torbjorn Sjöstrand, Stephen Mrenna, and Peter Z. Skands. "PYTHIA 6.4 Physics and Manual". In: *JHEP* 05 (2006), p. 026. DOI: 10.1088/1126-6708/2006/05/026. arXiv: hep-ph/0603175 [hep-ph].

[13] Torbjörn Sjöstrand et al. "An Introduction to PYTHIA 8.2". In: *Comput. Phys. Commun.* 191 (2015), pp. 159–177. DOI: 10.1016/j.cpc.2015.01.024. arXiv: 1410.3012 [hep-ph].

[14] Fabio Cascioli, Philipp Maierhofer, and Stefano Pozzorini. "Scattering Amplitudes with Open Loops". In: *Phys. Rev. Lett.* 108 (2012), p. 111601. DOI: 10.1103/PhysRevLett.108.111601. arXiv: 1111.5206 [hep-ph].

[15] Federico Buccioni, Stefano Pozzorini, and Max Zoller. "On-the-fly reduction of open loops". In: *Eur. Phys. J.* C78.1 (2018), p. 70. DOI: 10.1140/epjc/s10052-018-5562-1. arXiv: 1710.11452 [hep-ph].

[16] Gavin Cullen et al. "Automated One-Loop Calculations with GoSam". In: *Eur. Phys. J.* C72 (2012), p. 1889. DOI: 10.1140/epjc/s10052-012-1889-1. arXiv: 1111.2034 [hep-ph].

[17] Gavin Cullen et al. "G*OSAM*-2.0: a tool for automated one-loop calculations within the Standard Model and beyond". In: *Eur. Phys. J.* C74.8 (2014), p. 3001. DOI: 10.1140/epjc/s10052-014-3001-5. arXiv: 1404.7096 [hep-ph].

[18] Stefano Actis et al. "RECOLA: REcursive Computation of One-Loop Amplitudes". In: *Comput. Phys. Commun.* 214 (2017), pp. 140–173. DOI: 10.1016/j.cpc.2017.01.004. arXiv: 1605.01090 [hep-ph].

[19] S. Frixione, Z. Kunszt, and A. Signer. "Three jet cross-sections to next-to-leading order". In: *Nucl. Phys.* B467 (1996), pp. 399–442. DOI: 10.1016/0550-3213(96)00110-1. arXiv: hep-ph/9512328 [hep-ph].

[20] Rikkert Frederix et al. "Automation of next-to-leading order computations in QCD: the FKS subtraction". In: *Journal of High Energy Physics* 2009.10 (Oct. 2009), pp. 003–003. DOI: 10.1088/1126-6708/2009/10/003. URL: https://doi.org/10.1088%2F1126-6708%2F2009%2F10%2F003.

[21] Tomáš Ježo and Paolo Nason. "On the Treatment of Resonances in Next-to-Leading Order Calculations Matched to a Parton Shower". In: *JHEP* 12 (2015), p. 065. DOI: 10.1007/JHEP12(2015)065. arXiv: 1509.09071 [hep-ph].

[22] Jürgen Reuter et al. "Automation of NLO processes and decays and POWHEG matching in WHIZARD". In: *Journal of Physics: Conference Series* 762 (Oct. 2016), p. 012059. ISSN: 1742-6596. DOI: 10.1088/1742-6596/762/1/012059. URL: http://dx.doi.org/10.1088/1742-6596/762/1/012059.

[23] W. Kilian, J. Reuter, and T. Robens. "NLO Event Generation for Chargino Production at the ILC". In: *Eur. Phys. J.* C48 (2006), pp. 389–400. DOI: 10.1140/epjc/s10052-006-0048-y. arXiv: hep-ph/0607127 [hep-ph].

[24] T. Binoth et al. "Next-to-leading order QCD corrections to pp –> b anti-b b anti-b + X at the LHC: the quark induced case". In: *Phys. Lett.* B685 (2010), pp. 293–296. DOI: 10.1016/j.physletb.2010.02.010. arXiv: 0910.4379 [hep-ph].

[25] Nicolas Greiner et al. "NLO QCD corrections to the production of two bottom-antibottom pairs at the LHC". In: *Phys. Rev. Lett.* 107 (2011), p. 102002. DOI: 10.1103/PhysRevLett.107.102002. arXiv: 1105.3624 [hep-ph].

[26] Bijan Chokoufé Nejad et al. "NLO QCD predictions for off-shell $t\bar{t}$ and $t\bar{t}H$ production and decay at a linear collider". In: *JHEP* 12 (2016), p. 075. DOI: 10.1007/JHEP12(2016)075. arXiv: 1609.03390.

[27] Fabian Bach et al. "Fully-differential Top-Pair Production at a Lepton Collider: From Threshold to Continuum". In: *JHEP* 03 (2018), p. 184. DOI: 10.1007/JHEP03(2018)184. arXiv: 1712.02220 [hep-ph].

[28] G Peter Lepage. "A new algorithm for adaptive multidimensional integration". In: *Journal of Computational Physics* 27.2 (May 1978), pp. 192–203. DOI: 10.1016/0021-9991(78)90004-9. URL: https://doi.org/10.1016%2F0021-9991%2878%2990004-9.

[29] G P Lepage. *VEGAS - an adaptive multi-dimensional integration program*. Tech. rep. CLNS-447. Ithaca, NY: Cornell Univ. Lab. Nucl. Stud., Mar. 1980. URL: http://cds.cern.ch/record/123074.

[30] Ronald Kleiss and Roberto Pittau. "Weight optimization in multichannel Monte Carlo". In: *Computer Physics Communications* 83.2-3 (Dec. 1994), pp. 141–146. ISSN: 0010-4655. DOI: 10.1016/0010-4655(94)90043-4. URL: http://dx.doi.org/10.1016/0010-4655(94)90043-4.

[31] T. Ohl. "Vegas revisited: Adaptive Monte Carlo integration beyond factorization". In: *Computer Physics Communications* 120.1 (July 1999), pp. 13–19. DOI: 10.1016/s0010-4655(99)00209-x. URL: https://doi.org/10.1016%2Fs0010-4655%2899%2900209-x.

[32] F. James. "Monte Carlo theory and practice". In: *Reports on Progress in Physics* 43.9 (1980), p. 1145.

[33] Petros D. Draggiotis, Andre van Hameren, and Ronald Kleiss. "SARGE: An Algorithm for generating QCD antennas". In: *Phys. Lett.* B483 (2000), pp. 124–130. DOI: 10.1016/S0370-2693(00)00532-3. arXiv: hep-ph/0004047 [hep-ph].

[34] Andre van Hameren and Ronald Kleiss. "Generating QCD antennas". In: *Eur. Phys. J.* C17 (2000), pp. 611–621. DOI: 10.1007/s100520000508. arXiv: hep-ph/0008068 [hep-ph].

[35] Brian Gough. *GNU scientific library reference manual*. Network Theory Ltd., 2009.

[36] M. Beyer et al. "Determination of New Electroweak Parameters at the ILC - Sensitivity to New Physics". In: *Eur. Phys. J.* C48 (2006), pp. 353–388. DOI: 10.1140/epjc/s10052-006-0038-0. arXiv: hep-ph/0604048 [hep-ph].

[37] Ana Alboteanu, Wolfgang Kilian, and Juergen Reuter. "Resonances and Unitarity in Weak Boson Scattering at the LHC". In: *JHEP* 11 (2008), p. 010. DOI: 10.1088/1126-6708/2008/11/010. arXiv: 0806.4145 [hep-ph].

[38] Wolfgang Kilian et al. "High-Energy Vector Boson Scattering after the Higgs Discovery". In: *Phys. Rev.* D91 (2015), p. 096007. DOI: 10.1103/PhysRevD.91.096007. arXiv: 1408.6207 [hep-ph].

[39] Wolfgang Kilian et al. "Resonances at the LHC beyond the Higgs boson: The scalar/tensor case". In: *Phys. Rev.* D93.3 (2016), p. 036004. DOI: 10.1103/PhysRevD.93.036004. arXiv: 1511.00022 [hep-ph].

[40] Christian Fleper et al. "Scattering of W and Z Bosons at High-Energy Lepton Colliders". In: *Eur. Phys. J.* C77.2 (2017), p. 120. DOI: 10.1140/epjc/s10052-017-4656-5. arXiv: 1607.03030 [hep-ph].

[41] Alessandro Ballestrero et al. "Precise predictions for same-sign W-boson scattering at the LHC". In: *Eur. Phys. J.* C78.8 (2018), p. 671. DOI: 10.1140/epjc/s10052-018-6136-y. arXiv: 1803.07943 [hep-ph].

[42] Simon Brass et al. "Transversal Modes and Higgs Bosons in Electroweak Vector-Boson Scattering at the LHC". In: *Eur. Phys. J.* C78.11 (2018), p. 931. DOI: 10.1140/epjc/s10052-018-6398-4. arXiv: 1807.02512 [hep-ph].

[43] Jürgen Reuter and Daniel Wiesler. "Distorted mass edges at LHC from supersymmetric leptoquarks". In: *Phys. Rev.* D84 (2011), p. 015012. DOI: 10.1103/PhysRevD.84.015012. arXiv: 1010.4215 [hep-ph].

[44] Niklas Pietsch et al. "Extracting Gluino Endpoints with Event Topology Patterns". In: *JHEP* 07 (2012), p. 148. DOI: 10.1007/JHEP07(2012)148. arXiv: 1206.2146 [hep-ph].

[45] Jürgen Reuter and Daniel Wiesler. "A Fat Gluino in Disguise". In: *Eur. Phys. J.* C73.3 (2013), p. 2355. DOI: 10.1140/epjc/s10052-013-2355-4. arXiv: 1212.5559 [hep-ph].

[46] Gene 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). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560. URL: http://doi.acm.org/10.1145/1465482.1465560.

[47] John L. Gustafson. "Reevaluating Amdahl's Law". In: *Commun. ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415. URL: http://doi.acm.org/10.1145/42411.42415.

[48] William Gropp et al. *Using MPI & Using MPI-2*. Cambridge, MA, USA: MIT Press, 2014. ISBN: 9780262571340.

[49] Robit Chandra et al. *Parallel Programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN: 1-55860-671-8, 9781558606715.

[50] Robert W. Numrich. *Parallel Programming with Co-Array Fortran*. Boca Raton, FL, USA: CRC Press/Taylor & Francis, 2018. ISBN: 9781439840047.

[51] Pierre L'Ecuyer and Richard Simard. "Testu01: A C Library for Empirical Testing of Random Number Generators". In: *ACM Trans. Math. Softw.* 33.4 (Aug. 2007), 22:1–22:40. ISSN: 0098-3500. DOI: 10.1145/1268776.1268777. URL: http://doi.acm.org/10.1145/1268776.1268777.

[52] George Marsaglia. "The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness". In: (1995). URL: https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/.

[53] A. Wald and J. Wolfowitz. "On a Test Whether Two Samples are from the Same Population". In: *Ann. Math. Statist.* 11.2 (June 1940), pp. 147–162. DOI: 10.1214/aoms/1177731909. URL: https://doi.org/10.1214/aoms/1177731909.

[54] Robert G. Brown, Dirk Eddelbüttel, and David Bauer. In: (2018). URL: http://webhome.phy.duke.edu/~rgb/General/dieharder.php.

[55] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-89684-2.

[56] Martin Luscher. "A Portable high quality random number generator for lattice field theory simulations". In: *Comput. Phys. Commun.* 79 (1994), pp. 100–110. DOI: 10.1016/0010-4655(94)90232-1. arXiv: hep-lat/9309020 [hep-lat].

[57] Lev N. Shchur and Paolo Butera. "The RANLUX generator: Resonances in a random walk test". In: *Int. J. Mod. Phys.* C9 (1998), pp. 607–624. DOI: 10.1142/S0129183198000509. arXiv: hep-lat/9805017 [hep-lat].

[58] Pierre L'Ecuyer et al. "An Object-Oriented Random-Number Package with Many Long Streams and Substreams". In: *Operations Research* 50.6 (Dec. 2002), pp. 1073–1075. DOI: 10.1287/opre.50.6.1073.358. URL: http://dx.doi.org/10.1287/opre.50.6.1073.358.

[59] Message-Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1*. Stuttgart, Germany: High Performace Computing Center (HLRS), 2015.

[60] Richard Kreckel. "Parallelization of adaptive MC integrators". In: *Computer Physics Communications* 106.3 (Nov. 1997), pp. 258–266. DOI: 10.1016/s0010-4655(97)00099-4. URL: https://doi.org/10.1016%2Fs0010-4655%2897%2900099-4.

[61] Edward Boos and Thorsten Ohl. "Minimal gauge invariant classes of tree diagrams in gauge theories". In: *Phys. Rev. Lett.* 83 (1999), pp. 480–483. DOI: 10.1103/PhysRevLett.83.480. arXiv: hep-ph/9903357 [hep-ph].