EPJ Web of Conferences **245**, 02008 (2020) https://doi.org/10.1051/epjconf/202024502008
CHEP 2019

Using multiple engines in the Virtual Monte Carlo package

Benedikt Volkel[1,2,*], *Andreas* Morsch[2], *Ivana* Hřivnáčová[3], *Jan Fiete* Grosse-Oetringhaus[2], and *Sandro* Wenzel[2]

[1]Ruprecht-Karls-University Heidelberg, Germany
[2]European Organization of Nuclear Research (CERN), Geneva, Switzerland
[3]Université Paris-Saclay, CNRS/IN2P3, IJCLab, 91405 Orsay, France

Abstract. The Virtual Monte Carlo (VMC) package provides a unified interface to different detector simulation transport engines such as GEANT3 and GEANT4. It has been in production use in various experiments but so far the simulation of one event was restricted to the usage of a single chosen engine.
We introduce here the possibility to mix multiple engines within the simulation of a single event. Depending on user conditions the simulation is partitioned among the chosen engines, for instance to profit from each of their advantages or specific capabilities. Such conditions can depend on phase space, geometry, particle type or an arbitrary combination.
As a main achievement, this development allows for the implementation of *fast simulation* kernels at the VMC level which can be used stand-alone or together with full simulation engines. This capability is crucial to cope with largely increasing data expected in future LHC runs.

1 Introduction

The simulation of particles traversing complex detector geometries is one of the crucial building blocks of event simulation in heavy-ion and high-energy physics. Such simulations usually transport $O(100) - O(10000)$ primary particles per event. This requires frameworks capable of modelling the interactions with the detector materials and solving geometry propagation tasks. Commonly used packages are GEANT3 [2], GEANT4 [3–5] and FLUKA [6, 7].

To cover specific needs or to automate workflows, those transport codes are commonly utilised via additional experiment specific layers. The ALICE experiment at the LHC uses the Virtual Monte Carlo (VMC) [1] library which defines common interfaces and functionalities through abstract classes. In this way, the experiment's software framework does not depend on a specific transport engine[1].

In the previous implementation of VMC a single event was simulated with one chosen engine and it is not possible to dispatch to a fast simulation from the VMC code. The last point is an important limitation as their usage becomes more and more crucial especially in view of largely increasing data expected in the upcoming LHC runs.

*e-mail: benedikt.volkel@cern.ch

[1]The implementation for the FLUKA VMC interface has been restricted for the ALICE collaboration in 2010 and will not be discussed in this paper.

This paper introduces extensions to the VMC framework allowing for the *partitioning* of one event among multiple different engines. Such a partitioning could depend on particle type or phase space, detector geometry, other user defined conditions and as such it provides the capability to dispatch the transport to a VMC based fast simulation.

Sec. 2 outlines the basic VMC workflow. The central code developments and new interfaces are explained in Sec. 3 followed by proof-of-principle examples in Sec. 4. A final discussion and outlook is given in Sec. 5.

2 The VMC implementation and workflow

The backbone of the VMC package is built by the three abstract classes:

- `TVirtualMC`: Define and devise the interface to a transport engine backend. A concrete implementation of such an interface will be referred to as `VMCEngine` in the following.
- `TVirtualMCApplication`: Build the bridge between the running `VMCEngine` instance and the user logic or user code specific to a particular experiment. A derived user implementation will be referred to as `UserApp` in the following.
- `TVirtualMCStack`: Particle stack helper class to collect and access transported particles. A derived user implementation will be referred to as `UserStack` in the following.

In the multiple engine extension to the VMC package discussed here, two new classes are introduced:

- `TMCManager`: Registers and steers multiple engines and their particle stacks.
- `TMCManagerStack`: Particle stack helper class utilised by `TMCManager` to arrange multiple stacks internally and synchronise them with `UserStack`.

2.1 General considerations and workflow

The VMC workflow is sketched in Fig. 1 which is the same for both running with a single or multiple engines. The difference is only in how the run is steered which will be explained in section 3.1. *Start* marks the point when a simulation run is steered and *End* denotes when a run is fully finished. Each box represents a stage where a corresponding method of `UserApp` is called. This gives the possibility to inject additional user routines.

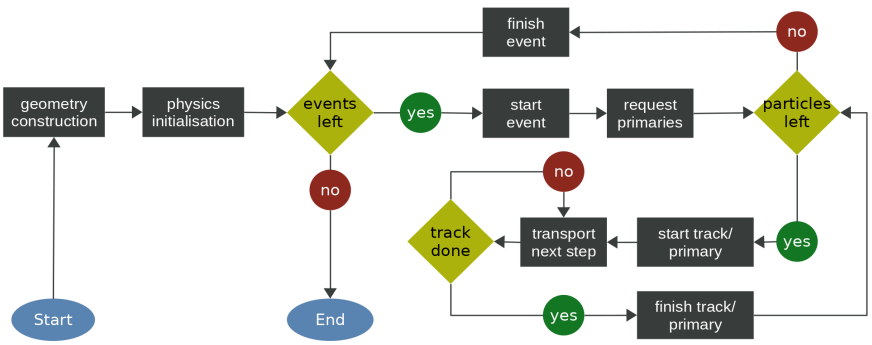

Figure 1. Sketch of the general VMC workflow. Boxes represent additional interaction between a `VMCEngine` and `UserApp`.

As the VMC framework makes use of different ROOT [8] classes, it also typically uses its system for geometry description and navigation. Currently, only geometries described via ROOT's `TGeoManager` are supported when running with multiple engines.

The developments described in this paper were made carefully to ensure backwards-compatibility for user-defined classes deriving from `TVirtualMCApplication` and `TVirtualMCStack`. Thus, there is no need to change anything concerning these user implementations when only a single engine is used.

3 Code developments and interfaces

This section introduces new classes, interfaces and extensions to allow for event partitioning between multiple engines. Fig. 2 sketches the implementation and interplay of the new classes `TMCManager` and `TMCManagerStack` in the context of a multiple engine run. In the following, key features are explained in more detail focussing on how a simulation run with multiple engines is handled and how existing user classes need to be adapted for that scenario.

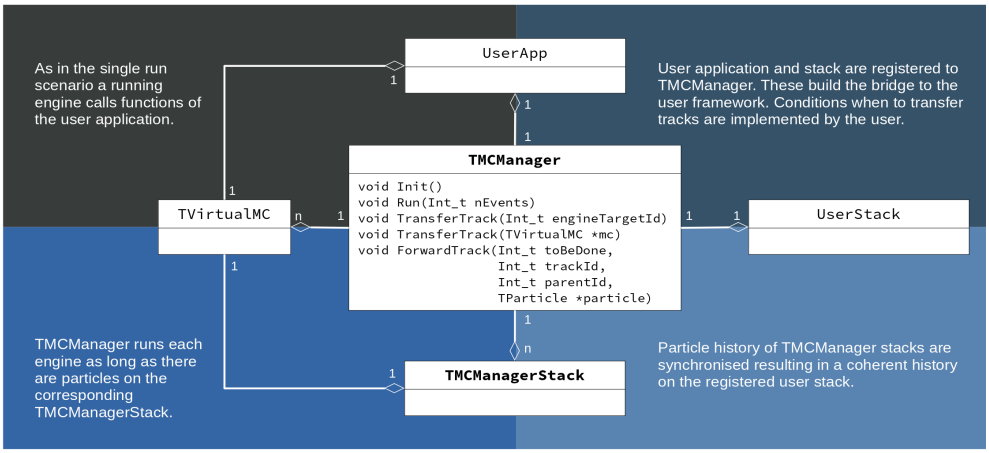

Figure 2. Sketch of the general VMC workflow. Boxes represent additional interaction between a `VMCEngine` and `UserApp`.

3.1 Introducing `TMCManager` and `TMCManagerStack`

Multiple engine support is enabled if the `TMCManager` singleton object is present. In that case `TMCManager::Instance()` returns a valid pointer and `null` otherwise. If it is not present, the system will run only with one engine, and will trigger an assertion if more than one `VMCEngine` is created.

When running with multiple engines, each `VMCEngine` registers itself to the manager object during construction. In addition, `UserApp` must be registered as well as the user stack. The latter will not be visible to the engines. Instead, each engine has a pointer to its individual `TMCManagerStack` object set by `TMCManager` which will synchronise and merge partial stack histories to one coherent history on the user stack.

A run with multiple engines is initialised and run calling `TMCManager::Init()` and `TMCManager::Run(Int_t nEvents)`, respectively. Thus, it follows the workflow of the single run where the initialisation and run are steered from `VMCEngine`.

Transferring tracks

A track is transferred to another engine via `TMCManager::TransferTrack(Int_t engineId)` or `TMCManager::TransferTrack(TVirtualMC *mc)` by passing the target engine's ID or a pointer to it. A possible use case is given in Listing 1 where a track is transferred if it enters the volume with the ID `volIdChange`. `TMCManager` interrupts the transport and caches the current track status as well as the navigator state. The latter is especially useful to re-initialise `TGeoNavigator` when the paused track is resumed. It avoids the time consuming task of searching the geometry tree again to find the current volume.

```
UserApp::Stepping()
{
    // Some implementation

    // Now make the decision whether the track should be transferred based on
     the current volume ID
    // (Need to pass a reference where the copy number is saved in addition.)
    Int_t copyNo;
    if(fMC->CurrentVolID(copyNo) == volIdChange) {
        // If target and current engine are the same, nothing will happen
        fMCManager->TransferTrack(targetEngineId);
    }
}
```

Listing 1. Transfer track during stepping based on volume ID.

Making the user stack ready for multiple engine usage

The user is responsible for recording, managing and indexing created tracks via a specific `UserStack` implementation. To comply with the user-specific stacking procedure, each attempt to push a track during simulation is always forwarded to the user stack. Listing 2 shows an example implementation of a `UserStack`. First, a track is created and indexed as if it was a run with only a single engine. Afterwards, the pointer to the track object is passed to `TMCManager::ForwardTrack(...)` along with its ID and parent ID. In general, no further changes are necessary to use a previously implemented `UserStack` with multiple engines.

```
UserStack::PushTrack(Int_t toBeDone, Int_t parent, ..., Int_t& ntr, ...)
{
    // Some implementation as they were done for a single run
    TParticle* particle = new TParticle(...);

    // Derive the ID of the track
    ntr = GetNewTrackID();

    // Some further implementation as they were done for a single run

    // Decide if multi-run by checking for valid pointer of a cached
    TMCManager pointer
    if(fMCManager) {
        // Forward to current engine...
        fMCManager->ForwardTrack(toBeDone, ntr, parent, particle);
    }
}
```

Listing 2. Forward track in `UserStack` when running with multiple engines.

Further important methods

- `void ConnectEnginePointer(TVirtualMC *&mc)`: Keep the passed pointer reference up-to-date to always point to the currently running engine.
- `TVirtualMC* GetCurrentEngine()`: Return a pointer to the current engine.
- `template <typename F> void Apply(F engineLambda)`: Apply a callable object `engineLambda`, which takes a pointer to a `TVirtualMC` object, to all registered engines.
- `Bool_t RestoreGeometryState()`: Set `TGeoNavigator` to the state the current track was paused at. Returns `kFALSE` if it could not be restored.
- `Bool_t RestoreGeometryState(Int_t trackId, Bool_t checkTrackIdRange = kTRUE)`: Set `TGeoNavigator` to the state of the track with ID `trackId` where it was paused at. If `checkTrackIdRange` is set to `kFALSE`, it has to be ensured that the track with that track ID exists. Returns `kFALSE` if the state could not be restored. This method is utilised by `TMCManager` itself and it restores necessary navigator states automatically.

3.2 Modification and extension of `TVirtualMCApplication` and `TVirtualMC`

A `UserApp` can automatically initialise a multi run scenario with the method `TVirtualMCApplication::RequestMCManager()` during construction which registers the application to the manager. The latter is then accessible via the member `TVirtualMCApplication::fMCManager` while the member `TVirtualMCApplication::fMC` always points to the currently running engine.

The previous singleton property of the `TVirtualMC` class has been lifted to allow for multiple instances of that type. To ensure backwards-compatibility, the static member `fgMC` as well as the static access method `GetMC()` returning that member are kept. When running with multiple engines, `fgMC` is updated by `TMCManager` whenever the engine changes.

Event processing

Having multiple engines, a track might have been transported already to its current position. To make an engine aware of that, the private virtual method `ProcessEvent(Int_t eventId, Bool_t isInterruptible)` has been introduced. Its implementation in a `VMCEngine` has to check whether a track was already transported partially and if so, any procedures related to starting a new track must be omitted. This method is solely used by `TMCManager` which is a *friend class* of `TVirtualMC`.

If a track meets the conditions to be transferred to another engine, it cannot simply be stopped by `StopTrack()` as this calls `TVirtualMCApplication` methods related to finishing a track. This is achieved by the private virtual method `InterruptTrack()`. Its implementation in a `VMCEngine` has to make sure that the track is not transported further but at the same time any procedures related to finishing a track must be omitted.

GEANT3_VMC and GEANT4_VMC

The necessary pure virtual methods have been implemented for the VMC interfaces to GEANT3 and GEANT4. Most importantly, the interaction with the particle stack and the geometry navigation has been revisited and modified accordingly. Both interfaces can now handle the scenario when a track has already been transported partially and if so, `TMCManager` is requested to to restore the corresponding geometry state. GEANT3 and GEANT4 are therefore ready to be used in a multiple engine scenario in the VMC framework.

4 Proof-of-principle example

Two proof-of-principle examples are presented, the first one mixing the full simulation engines GEANT3 and GEANT4 and the second mixing a custom fast-simulation-like VMC engine with GEANT4[2].

4.1 Mixing GEANT3 and GEANT4 simulation

The left sketch in Fig. 3 shows a vanilla sampling calorimeter with passive (dark) and active (red) layers which will be called *ABSO* and *GAPX*, respectively. GEANT3 is responsible for the simulation of the ABSO layers whereas GEANT4 takes care of the GAPX layers. The right plot in Fig. 3 shows the time elapsed in simulation for different simulation scenarios normalised to a GEANT3-only scenario. The horizontal axis shows the number of calorimeter layers while the overall thickness is fixed. Due to a more detailed simulation of GEANT4 this takes longer when compared to GEANT3, as expected. When partitioning the simulation as described above (green line), the simulation time reduces significantly compared to using only GEANT4. A flat curve indicates that the simulation time does not scale with the number of layers and hence not with the number of track transfers. Pausing and resuming tracks/engines do therefore not introduce any runtime overhead[3].

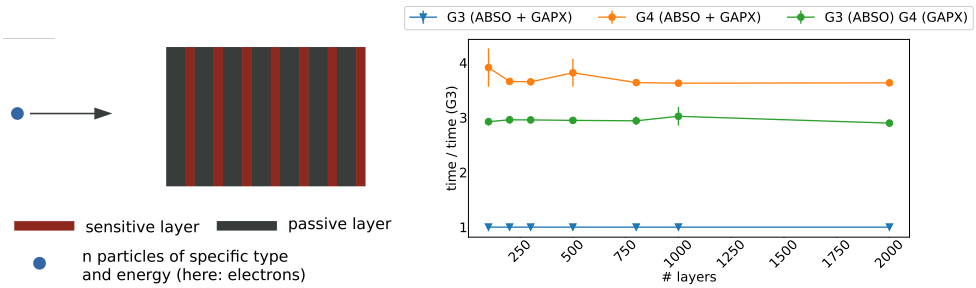

Figure 3. Sketch of a vanilla sampling calorimeter *(left)* and simulation time relative to the time elapsed for GEANT3 when using different engines *(right)*. Mixing the detailed GEANT4 simulation with GEANT3, the simulation time reduces significantly compared to using GEANT4 only.

4.2 Mixing GEANT4 with a custom fast-simulation-like VMC engine

The same geometry setup introduced in the previous sub-section is used to demonstrate the partitioning between GEANT4 and a custom fast-simulation-like engine[4]. GEANT4 transports the particles within the world volume up to the calorimeter and as soon as that is reached, the custom engine takes over. The latter parametrises the entire energy deposit in the calorimeter. To do so, an energy distribution was first simulated with GEANT4 which the fast-simulation-like engine draws the values from. The result can be seen in Fig. 4 showing the energy distributions obtained from the full simulation and the one from the custom engine.

[2]GEANT4 implements a fast simulation framework natively. However, those implementations cannot be used with other full simulation engines and therefore, this scenario is not considered here.

[3]These scenarios are available as example *E03c* in the GEANT4_VMC repository at https://github.com/vmc-project/geant4_vmc/tree/master/examples/E03/E03c.

[4]The corresponding implementations can be found at https://github.com/benedikt-voelkel/VMCFastSim and https://github.com/benedikt-voelkel/FastShower

The results are compatible, however, when using the fast-simulation-like engine, execution time reduces by one order of magnitude. Such a reduction of resource demands is one of the main purposes of fast simulation implementations.

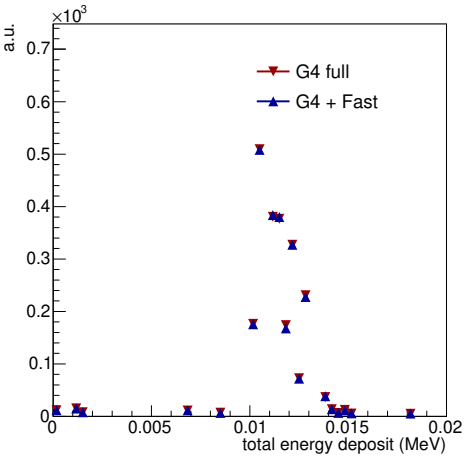

Figure 4. Total energy distribution obtained from a GEANT4 full simulation and a fast-simulation-like VMC implementation drawing from the full simulation distribution.

5 Conclusion

The VMC framework has been extended to allow running multiple transport engines such that the simulation of an event can be partitioned among those. The engines' particle stacks are managed automatically and synchronised with the user stack ensuring a coherent history.

It has been ensured that the previous scenario with only one engine is conserved and no changes are needed in previous user code in that case. In order to use those with multiple engines, the required changes are minimal and straightforward to implement.

Partitioning a simulation has been demonstrated in two examples, one mixing the full simulation engines GEANT3 and GEANT4 and another one mixing GEANT4 with minimal custom implementation of `TVirtualMC`. The latter can be seen as an example of how to incorporate fast simulation on the level of the VMC package and it can be combined with other interface from `TVirtualMC`.

References

[1] Hřivnáčová, I. et al., Proceedings of Computing in High Energy and Nuclear Physics, pp THJT006 (2003)

[2] Brun, R. et al, *GEANT3: user's guide Geant 3.10, Geant 3.11* (CERN, Geneva, 1987) Report number CERN-DD-EE-84-01

[3] Agostinelli, S. et al, Nucl. Inst. & Meth. in Phys. Res. A **506 no. 3**, 250–303 (2003)

[4] Allison, J. et al, Nucl. Inst. & Meth. in Phys. Res. A **835**, 186–225 (2016)

[5] Allison, J. et al, IEEE Transactions on Nuclear Science **53 no. 1**, 270–278 (2006)

[6] Ferrari, A. et al, *FLUKA: A multi-particle transport code* (CERN, Geneva, 2005) Report Number CERN-2005-010; INFN-TC-2005-11; SLAC-R-773

[7] Böhlen, T.T. et al, Nuclear Data Sheets **120**, 211–214 (2014)

[8] Rene Brun and Fons Rademakers, Nucl. Inst. & Meth. in Phys. Res. A **389**, 81–86 (1997) (see also https://root.cern.ch)