CMS Experience with Current Generators

Josh Bendavid (Caltech)

September 23, 2016
Beyond Leading Order Calculations on HPCs
FNAL

Introduction

- In considering future evolution of generator tools and increased use of HPC resources, useful to understand **how** we are using external generator programs in CMS at both a physics and technical level

CMS Software Overview

- Main CMS software application: CMSSW
- Modular C++ application which can be used for event generation, detector simulation, reconstruction, and analysis
- Configuration of CMSSW runs is steered with python-based configuration files
- Input and output with root-based EDM files, which store run-level, lumi-section-level (23s periods for real data), or event-level data products
- CMSSW links directly to many **externals**, externally maintained C, C++, fortran, or python software which is either an indirect dependency or is directly called from within CMSSW
- Externals are compiled with the same common libraries, compiler version as CMSSW and packaged together with a given release, starting from either a tarball from the author's website, from GENSER, or from a cms-managed github mirror

CMS Production Overview

- Python-based tools manage large-scale submission of CMSSW jobs to grid resources for central production of Monte Carlo, data processing, etc
- Jobs are assumed to be CMSSW jobs configured by the corresponding python-based configuration
- All input and output are assumed to be EDM files (with a few special cases)
- A similar mechanism is available to end users to submit analysis jobs
- CMSSW software and corresponding externals is made available on worker nodes through CVMFS (distributes http-based read-only filesystem)

CMS Software: Event Generation

- Basic paradigm: A C++ module with a common interface makes the needed calls to a linked external generator code in order to produce for each event a HepMC::GenEvent, which can then directly stored in the EDM output
- Configuration of the generator takes place within the CMSSW python configuration
- **Advantages:**
    - Uniform configuration and IO mechanism (production tools only have to deal with CMSSW)
    - No intermediate files needed (HepMC::GenEvent is passed along in memory to standard CMSSW/root IO mechanisms or directly to GEANT, which is also called from inside CMSSW)
- **Disadvantages:**
    - Each generator needs a dedicated interface needed in CMSSW and needs to be packaged as a CMSSW external
    - Initialization and event generation calls must be possible from within a C++ application
- In practice, Pythia, Herwig, Sherpa fit very nicely into this paradigm (with some preference for C++-based versions)

Example CMSSW GEN Configuration Fragment

```
import FWCore.ParameterSet.Config as cms

from Configuration.Generator.Pythia8CommonSettings_cfi import *
from Configuration.Generator.Pythia8CUEP8M1Settings_cfi import *

generator = cms.EDFilter("Pythia8GeneratorFilter",
  maxEventsToPrint = cms.untracked.int32(1),
  pythiaPylistVerbosity = cms.untracked.int32(1),
  filterEfficiency = cms.untracked.double(1.0),
  pythiaHepMCVerbosity = cms.untracked.bool(False),
  comEnergy = cms.double(13000.0),

  crossSection = cms.untracked.double(1.92043e+07),

  PythiaParameters = cms.PSet(
    pythia8CommonSettingsBlock,
    pythia8CUEP8M1SettingsBlock,
    processParameters = cms.vstring(
      'HardQCD:all = on',
      'PhaseSpace:pTHatMin = 50  ',
      'PhaseSpace:pTHatMax = 80  ',
    ),
    parameterSets = cms.vstring('pythia8CommonSettings',
                                'pythia8CUEP8M1Settings',
                                'processParameters',
                               )
  )
)
```

CMS Software: LHE Input

- CMS maintains its own LHE parser (based on xerces-c xml library)
- An LHE file can be read as input to a CMSSW job and is converted on the fly to C++ classes LHERunInfoProduct and LHEEventInfoProduct which store the relevant information and can be stored/read from EDM files (support for per-event weights added to CMS lhe parser and classes)
- LHE information can be passed as input to a hadronizer as part of the event generation step in CMSSW (using for example the Pythia8::LHAup mechanism to pass the needed information on the fly in memory)
- Advantage: Uniform hadronizer-independent storage and access to lhe information
- Disadvantage: We have to maintain our own lhe parser
- More recently Pythia8 (and soon Herwig7) LHE parsers can also be used as an alternative

LHE Input for Central Production

- CMS production tools do not work transparently with ascii LHE input (metadata not automatically available in data management system, skipping of events is inefficient, etc)
- It is possible to use privately produced LHE files for central production (user copies the files to eos and then a conversion step is run to produce EDM files containing the LHE products, which can then be used for further production steps for hadronization, simulation, etc)
- Disk space, file corruption, etc, are major issues when dealing with large sets of lhe files in this way
- **Common lesson for HPC usage:** Reduce/eliminate use of text files in workflows, or make them transient such that they can be written only to local SSD or ramdisk

Central production of LHE events

- LHE generators like Madgraph_aMC@NLO, POWHEG, etc cannot be directly called from within CMSSW in general
- Solution is "externalLHEProducer" a C++ CMSSW module which calls an external script, then reads the resulting LHE file (with the CMS lhe parser) and produces the necessary LHERunInfoProduct/LHEEventInfoProduct which can be stored in the EDM file and/or passed along to the hadronizer
- Further issue: LHE generator code cannot easily be included with CMSSW as an external, since each process requires dedicated (and sometimes dynamically generated) libraries
- Solution: "gridpacks" with pre-generated/compiled code, and with initial phase space integration results stored in a tarball
- Gridpacks are put in CVMFS and can be accessed by jobs (gridpack location is a configuration parameter of the externalLHEProducer module)
- Minimal and compact external input, and compressed EDM output make very large scale LHE production possible.
- CMS has produced $>$ 30 billion LHE events through this mechanism for Run 2 so far (trivial parallelization on the grid).

Gridpacks

- General gridpack mechanism used in CMS is modeled on the built-in functionality for LO processes in Madgraph_aMC@NLO
- We maintain scripts for Madgraph_aMC@NLO (including NLO processes), POWHEG, JHUGen to produce gridpack tarballs based on the appropriate input cards
- Important considerations:
  - Compiling code on batch workers is discouraged (should be possible to fully precompile everything)
  - Long initialization time for event generation is discouraged
  - Gridpack size is an issue (more than about 500MB for the tarball or 5GB decompressed starts to become problematic)
  - Gridpack generation step needs reliability and reasonable run-time "as the physicist waits" (we can use multi-core machines and/or condor/lsf batch queues to do the phase space integration, but does no good if process is bottle-necked by single-threaded steps, or individual long-running jobs)
- Similar mechanism used for Sherpa "Sherpacks" and in progress for Herwig7 NLO processes

Gridpacks

- Factorization of event generation into preparation stage and event generation likely an effective paradigm going forward
- Event generation is trivially parallelizable and well suited for conventional grid resources
- Phase-space integration for complex processes good candidate for HPC resources
- Effective full-scale use requires user-friendly procedure for gridpack preparation which can be widely used across the collaboration
- (Ideal World: User runs a script at CERN or at their Tier-2 which handles submission of their gridpack preparation to appropriate HPC resource authenticated by grid certificate)

Madgraph_aMC@NLO Experience

- Broadly used in CMS for a large number of processes at both LO and NLO
- Mainly using LSF batch system at CERN (common denominator for all collaborators)
  - Issue: Madgraph lsf support relies on shared filesystem. CERN afs system too slow/unreliable/small to effectively use for this purpose
  - CMS uses modified lsf cluster support to avoid shared filesystem (rsync transfers between master and child nodes)
  - **Reliable job retry logic essential**
- Madgraph uses statically linked executables (one set for each subprocess)
  - HUGE size of compiled code O(10 GB) for DY+0,1,2 jets NLO
  - Use lzma compression with very large dictionaries when preparing tarballs to deduplicate statically linked code in each subprocess
- Generation of diagrams/code was single threaded with memory bottlenecks in previous versions$\rightarrow$ multithreaded approach based on python multiprocess and pickled intermediate objects to disk

POWHEG

- POWHEG usage in CMS has mostly been with “simple” NLO processes, since number of processes with extra jets and MINLO is limited
- Limited use of parallelization in this case (mostly gridpacks easily produced on one machine)
- Adaptation of CMS scripts to (batch system)-based parallelization for more complex processes recently completed and in testing

Sherpa

- Use model:
  - Standalone run up to phase space integration results, packaged in Sherpack (analogous to gridpacks for Madgraph/POWHEG)
- MPI parallelization model has ironically been a limitation for CMS due to more limited available facilities (and limited people who can run there!) $\rightarrow$ strong motivation for HPC resources, but need to ensure they can ultimately be broadly utilized
- Past (albeit temporary) reliance on non-open-source libraries is very bad both for CMS software model and any adaptation to HPC (need to be able to recompile!)
- More details in CMS contribution to Sherpa users meeting in Jan. `https://indico.cern.ch/event/472944/contributions/1993254/attachments/1208554/1761980/lenzip_07012016.pdf`

Parameter Scans

- Parameter scans implemented in CMS by configuring jobs with a **list of generator configurations and/or gridpacks with associated weights**, provided through the same python-based mechanism
- Example config: `https://github.com/cms-sw/cmssw/blob/CMSSW_8_0_X/GeneratorInterface/Pythia8Interface/test/randomizedParametersSLHALHE.py`
- Configuration and/or gridpack choice is randomized every 200 events (tradeoff between re-initialization overhead and size of statistical fluctuations)
- Missing events from failed jobs are randomly distributed across parameter space $\rightarrow$ **no holes from failed jobs**

Conclusions

- Generator tools need to be carefully integrated into our software and production framework to be useful for large-scale usage
- This process is driven by important constraints on both the CMS side and in the generator tools
- Discussing these issues and technical details can hopefully lead to better and more effectively used generator tools and interfaces
- Equally relevant for (eventual) integration of HPC resources into broad-base production