SHERPA: Scaling on HPCs

Marek Schönherr*

Universität Zürich

CERN 16 Mar 2017

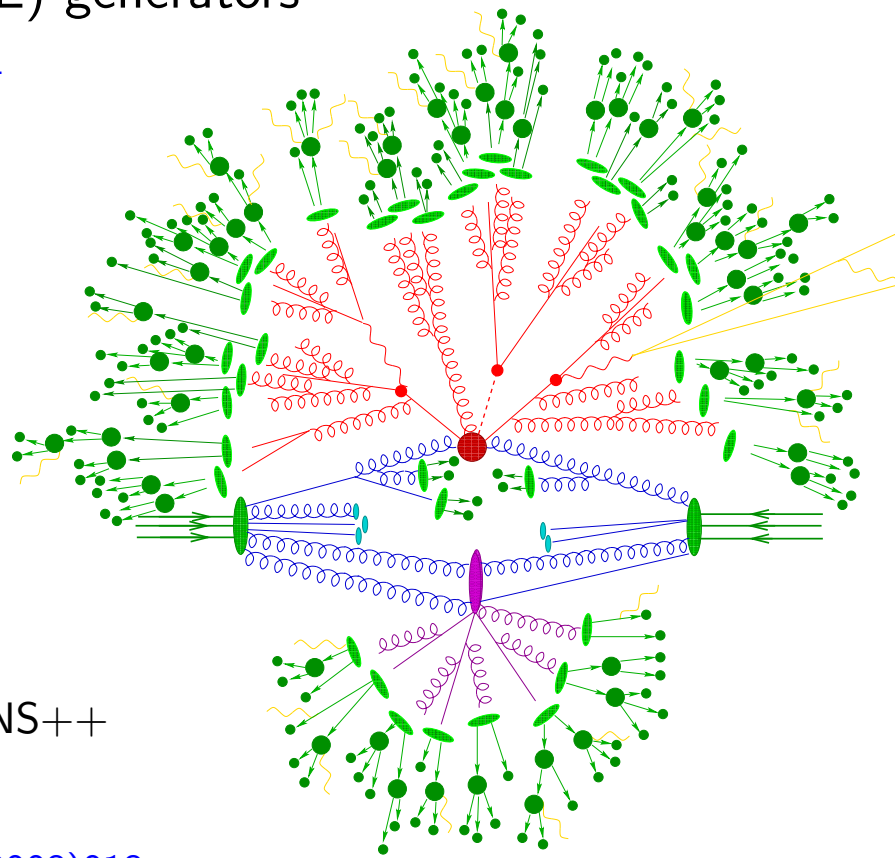
Universität Zürich[UZH]

*huge amounts of input from S. Höche

The SHERPA event generator framework

JHEP02(2009)007

- Two multi-purpose Matrix Element (ME) generators
  AMEGIC++ JHEP02(2002)044, EPJC53(2008)501
  COMIX JHEP12(2008)039, PRL109(2012)042001
- Two Parton Shower (PS) generators
  CSSHOWER JHEP03(2008)038
  DIRE EPJC75(2015)461
- A multiple interaction simulation
  AMISIC++ hep-ph/0601012
- A cluster fragmentation module
  AHADIC++ EPJC36(2004)381
- A hadron and τ decay package HADRONS++
- A higher order QED generator using
  YFS-resummation PHOTONS++ JHEP12(2008)018



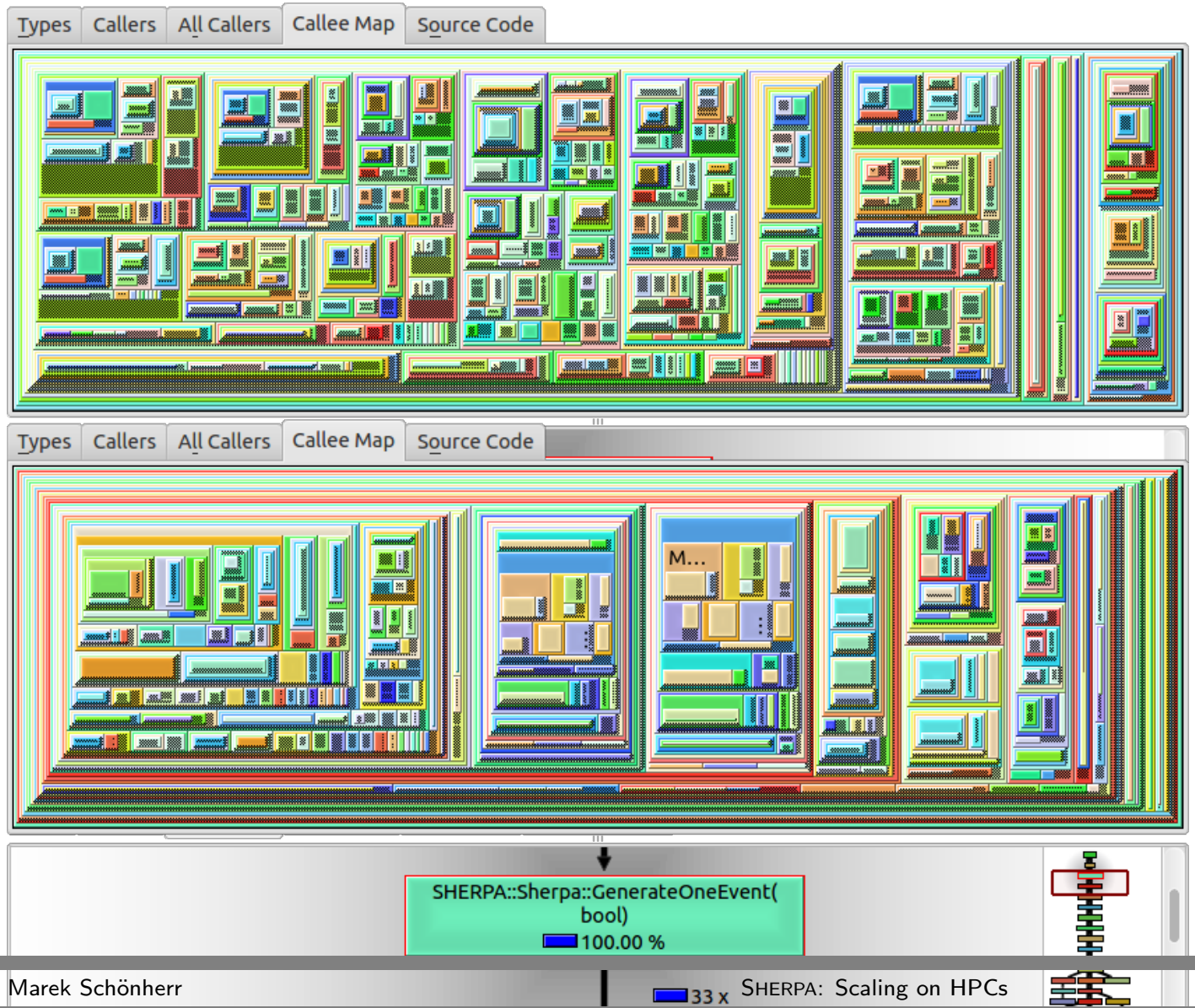
SHERPA is a full particle level event generator at the highest perturbative precisions: MEPS@NLO, NNLOPS

Code structure

Modularity

- SHERPA core for event handling and stearing
  comprises interfaces to each physics aspect
- each physics module is loaded at run time and is interchangeable
  Examples:
  - 2 matrix element generators
  - 2 parton showers
  - 2 hadronisation modules
- allows for easy extendability and user customisation
  Examples:
  - interface different loop generator
  - use user-supplied scale setting routines or phase space cuts
- modules are loaded when needed
- modules are added without modification to the main code

The caveat of object oriented code



← Initialization

← Event generation

Lessons from Mira (Argonne) - A sampler

I/O - The usual suspect

- Sherpa pre-2.2.0 performs thousands of file operations at startup (hadron decay information, process information & libraries, integrators, ...)
  causing distress on Mira due to the large number of concurrent processes (file system access is serial)
- 1^{st} attempt: Combine all process libraries into one – no luck
- 2^{nd} attempt: Combine info files into SQLite databases – no luck
- 3^{rd} attempt: Remove unnecessary stat calls – getting there
- 4^{th} attempt: Clean up code – better, but still not optimal

Lessons from Mira (Argonne) - A sampler

Code design

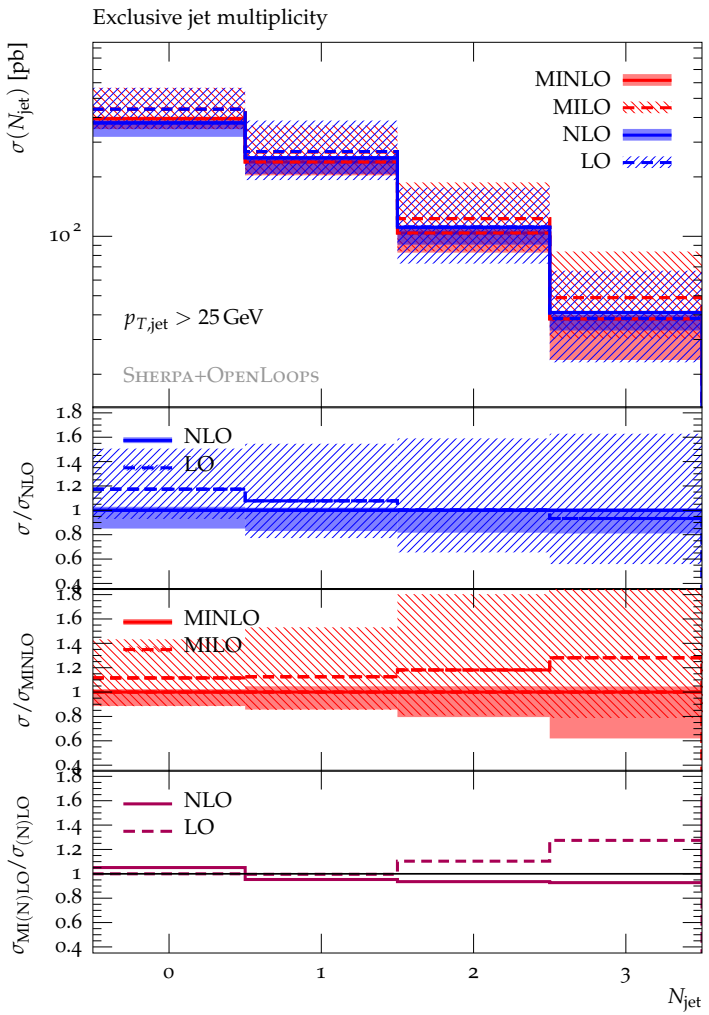
- Automated codes are stupid, ME generators in particular
- You can afford being stupid once, but not a million times over
- 1^{st} attempt: Redesign dipole mapping in Amegic – better
- 2^{nd} attempt: Redesign process initialization in Sherpa – even better
  Example on E5-2699: W+7j (all) 1.4GB, 1.5d to init, 15s to start
  W+8j (all) 5.9GB, 14d to init, 74s to start

Sherpa is a classic example of sacrificing speed for flexibility and maintainability.

$t\bar{t}$ + 3 jets at NLO

Höche,Maierhöfer,Moretti,Pozzorini,Siegert arXiv:1607.06934

- First computation of $t\bar{t}$+3 jets at NLO / MINLO accuracy
- Sherpa NLO MC framework using Comix Höche,Gleisberg arXiv:0808.3674 combined with OpenLoops Cascioli,Maierhöfer,Pozzorini arXiv:1111.5206
- Public results in NTuple format à la BlackHat collaboration arXiv:1310.7439 for easy analysis & recycling available at NERSC
- Scale dependence studied using $H_{T,m} = \sum m_\perp$ and MINLO Hamilton,Nason,Zanderighi arXiv:1206.3572 extended to massive partons

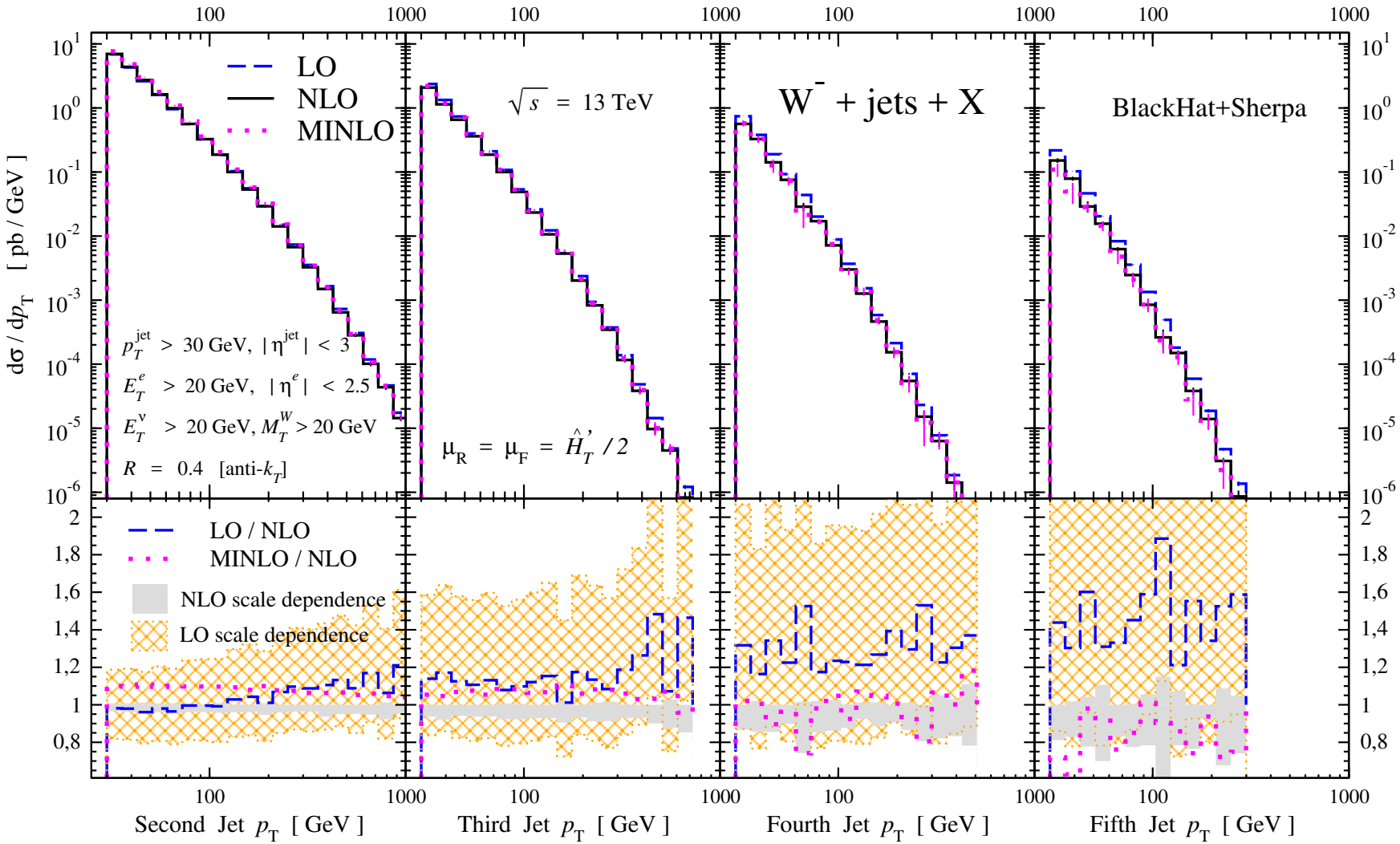


$t\bar{t}$ + 3 jets Performance on Bullet Cluster (SLAC)

Type	RAM/core [GB] init/run	Init Time	MC error	# Cores & Time
I gg→3g	85/127MB	20s	0.27%	64 × 4.5h
I qq→3g	68/110MB	13s	0.34%	64 × 1.1h
I gq→2g1q	72/114MB	17s	0.46%	64 × 58m
I gg→1g2q	48/90MB	10s	0.34%	64 × 31m
I qq→1g2q	150/187MB	43s	0.42%	64 × 3.2h
I gq→3q	87/126MB	25s	0.29%	64 × 1.3h
RS 2g→4g	501/574MB	4m 33s	1.4%	128 × 20.5h
RS 2q→4g	548/614MB	5m 21s	2.4%	128 × 7.7h
RS 1g1q→3g1q	553/621MB	4m 39s	0.88%	128 × 10.7h
RS 2g→2g2q	248/315MB	1m 29s	1.1%	128 × 4.3h
RS 2q→2g2q	1.9/1.9GB	12m 25s	1.1%	128 × 18.8h
RS 1g1q→1g3q	670/737MB	4m 41s	0.65%	128 × 1.08d
RS 2g→4q	208/274MB	1m	0.45%	128 × 4.5h
RS 2q→4q	1.8/1.8GB	9m 59s	0.61%	128 × 1.5d

Disclaimer: This does not include the costs for NTuple production

$W + 5$ jets at NLO



$W + 5$ jets Performance on Cori (NERSC)

Type	RAM/core [GB] init/run	Init Time	MC error	# Cores & Time	# Cores per Node
l qq	5.0/4.1GB	35m	1.2%	512 × 1d	28
l gq	1.4/1.2GB	11m	0.6%	512 × 1.8d	32
l gg	0.3/0.2GB	2m	0.9%	512 × 3.5h	32
RS qq→6g	1.4/1.1GB	30m	1.9%	1024 × 8h	32
RS gq→5g1q	1.3/1.1GB	25m	2.5%	1024 × 16h	32
RS gg→4g2q	0.7/0.6GB	12m	2.9%	1024 × 4.5h	32
RS qq→4g2q	16/12.3GB	6h	10.5%	16 × 13d	9 *
RS gq→3g3q	6.2/4.8GB	1.7h	4.3%	1024 × 1.8d	23
RS gg→2g4q	1.3/1.0GB	24m	2.8%	512 × 16h	32
RS qq→2g4q	32/24.4GB	12h	22%	16 × 9d	4 *
RS gq→1g5q	6.9/5.4GB	2h	34%	512 × 42h	16
RS gg→6q	1.5/1.2GB	30m	6.7%	512 × 23h	32
RS qq→6q	23/16.5GB	4h	35%	16 × 12d	7 *

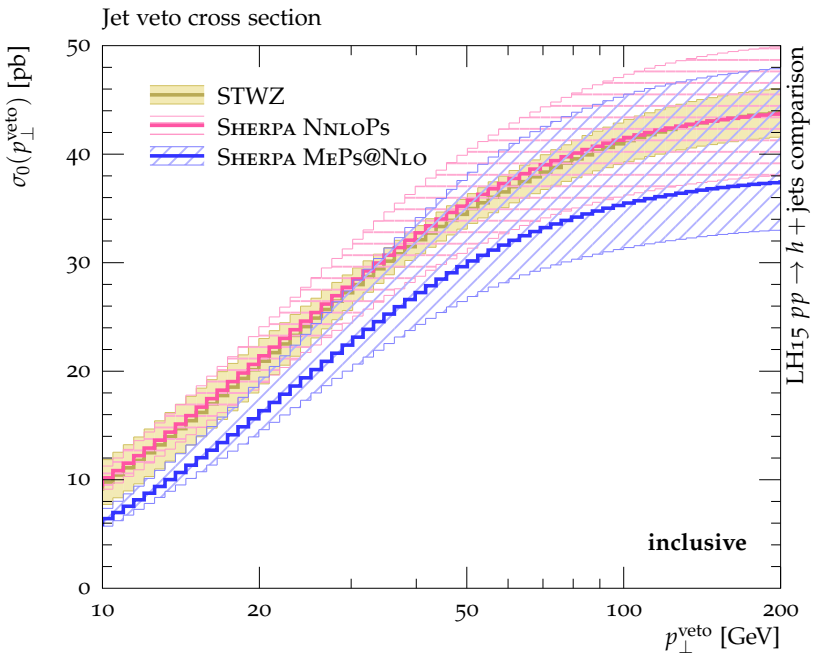
Disclaimer: This does not include the costs for NTuple production

* Computed on UCLA Hoffman 2 Cluster

$pp \to h +$ jets at MEPS@NLO and NNLOPS

- realsitic calculation
- including parton showering
- MEPS@NLO:
  $pp \to h + 0, 1, 2, 3j$ @ NLO
  incl. xs @ NLO
- NNLOPS:
  $pp \to h$ @ NNLO
  incl. xs @ NNLO
- Loops from GoSam and dedicated libs

LH'15 arXiv:1605.04692



$\Rightarrow$ **type of simulations of (most) interest to ATLAS**

Typical calculation for ATLAS

Not $pp \to W + 5$ jets

MEPS@NLO: $pp \to W + 0, 1, 2j\text{@NLO}, 3, 4, 5j\text{@LO} + X$

- processes of **very different complexities** simultaneously
- high jet multiplicities challenging both in the number of diagrams per process, the number of processes, and the number of potential clusterings per process (for scale setting and Sudakov rejections)
- large executables

NNLOPS: $pp \to W + X$

- needs many events for stability
- simple process and analytic (non-automatically generated) libs
- small executables

Typical calculation for ATLAS

Involved chain of modelling of physics processes:

- beam fragmentation
- hard interactions (ME)
- parton showering
- multiple interactions
- hadronisation
- hadron decays
- QED bremsstrahlung

with a priori variable and unpredictable program flows

$\rightarrow$ this is not just doing the same calculations over and over again

Parallisation paradigms

Sherpa is run in two steps:

1. **Integration:** pure ME calculation
   → independent evaluations that need to cross talk only to optimise VEGAS grids,
   → bottleneck lies in the use of stratified sampling

2. **Event generation:** completely independent calculations with arbitrary and unpredictable program flow
   → best trivially parallelised for minimum overhead

⇒ **bottleneck is the potential size of the executable**
can be solved using multithreading

ME integration

We need to know:

- the cross section of each partonic channel
- the optimal VEGAS weights and maximum of the remaining weight distribution for efficient unweighting

We have:

- highly stable and predictable program flow
  - always do the exact same calculation, only parameters (momenta, scales, coupling constants) vary

$\Rightarrow$ standard case for parallelisation using MPI,
multithreading (limited to cores on each node) reduces memory req.

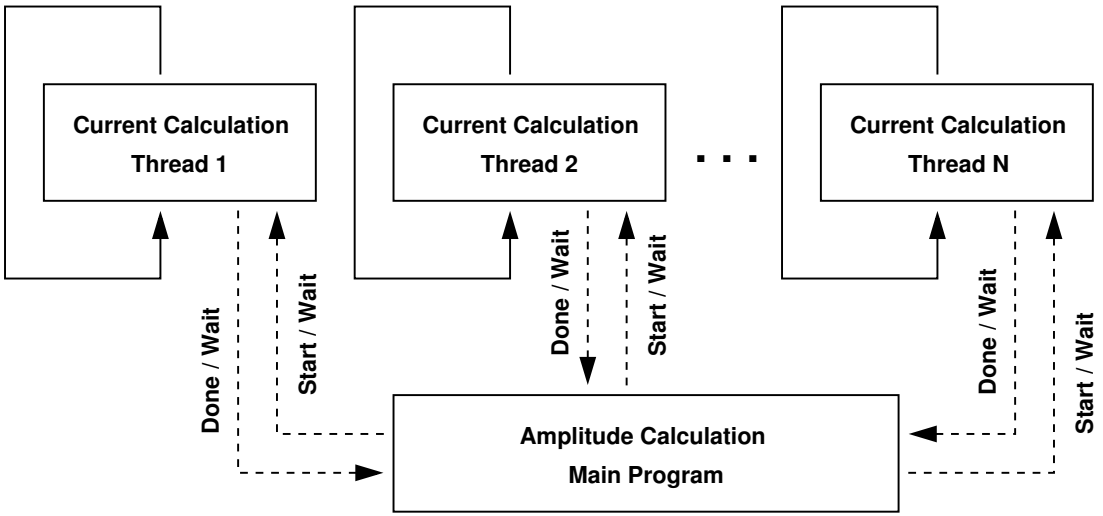
$\Rightarrow$ combine both

ME integration – multithreading

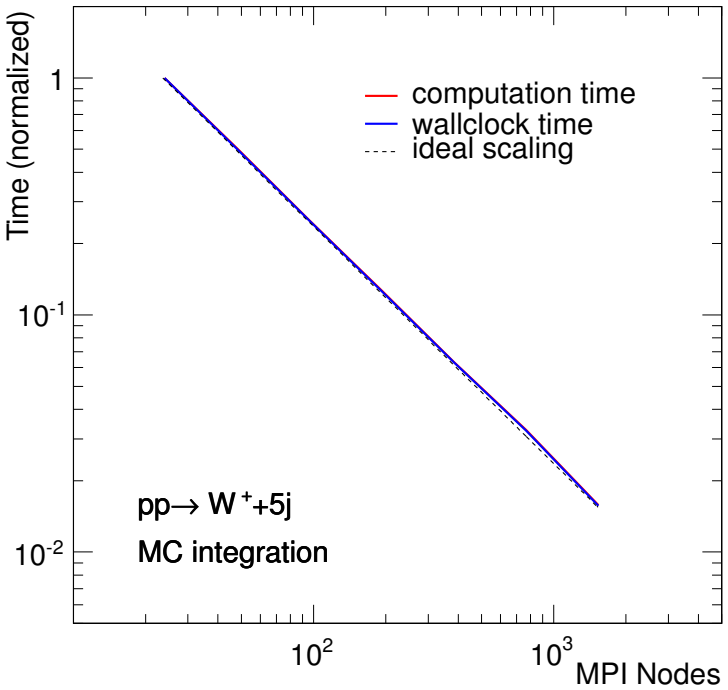
Natural application of threading: Recursive relations

- N-point currents in Berends-Giele recursion only depend on M-point currents, where $M < N$
- Easily parallelizable by breaking calculation into blocks which handle equal number of N-point currents

Gleisberg,Höche arXiv:0808.3674, Giele,Stavenga,Winter arXiv:1002.3446



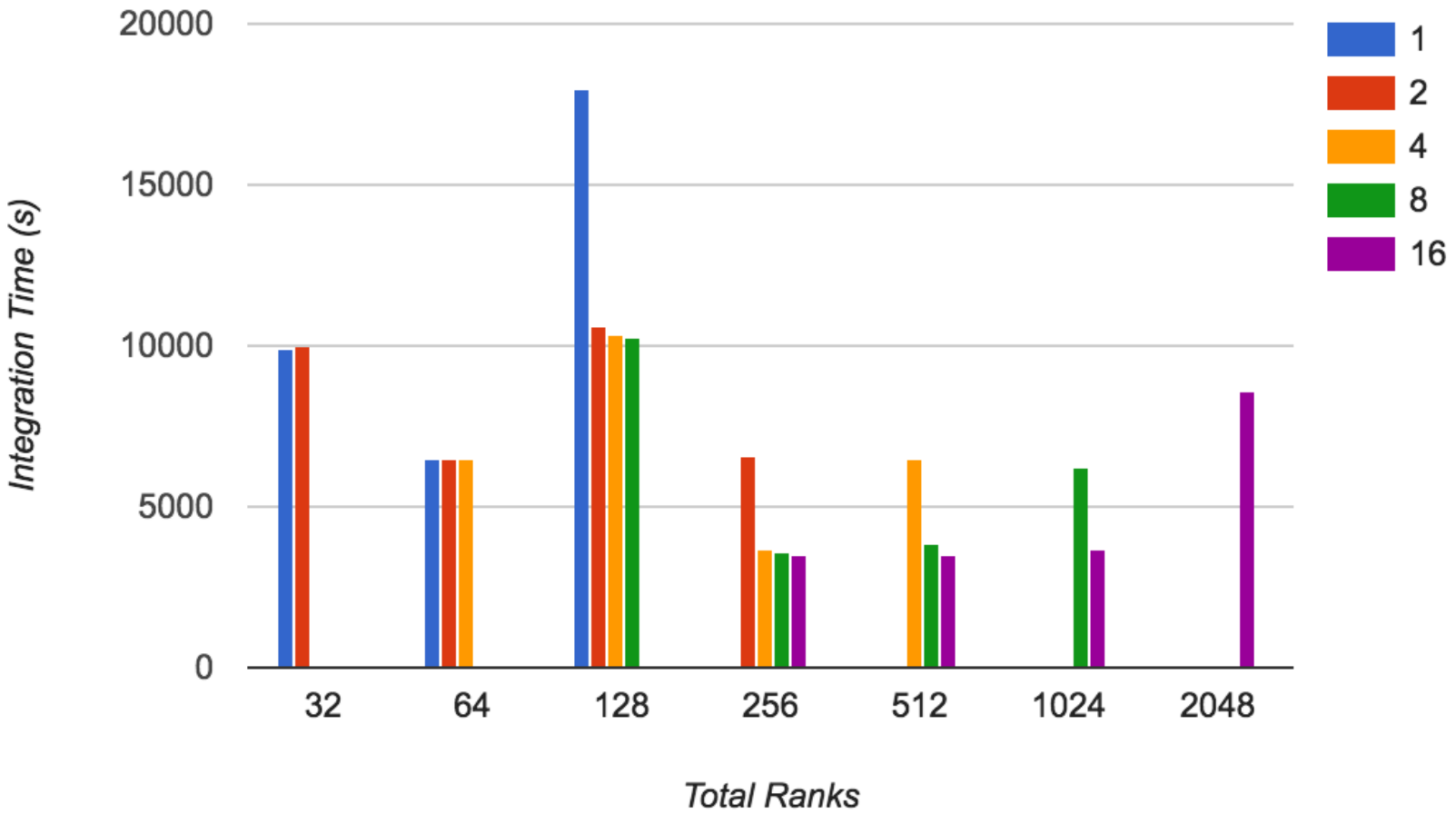
ME integration – scaling



- ideal scaling up to $\sim$ 1000 cores
- limited by the complexity of the process
  - need N points per iteration for VEGAS
  - $\frac{N}{n}$ points on each of the n cores
  - $t(\frac{N}{n}$ points$) \gg t$(MPI comm) for good scaling behaviour
  - MPI comm to master is $\sim$ serial
- example: 500k points per iteration on 10k cores leaves only 50 points per core, while MPI comm time has increased significantly

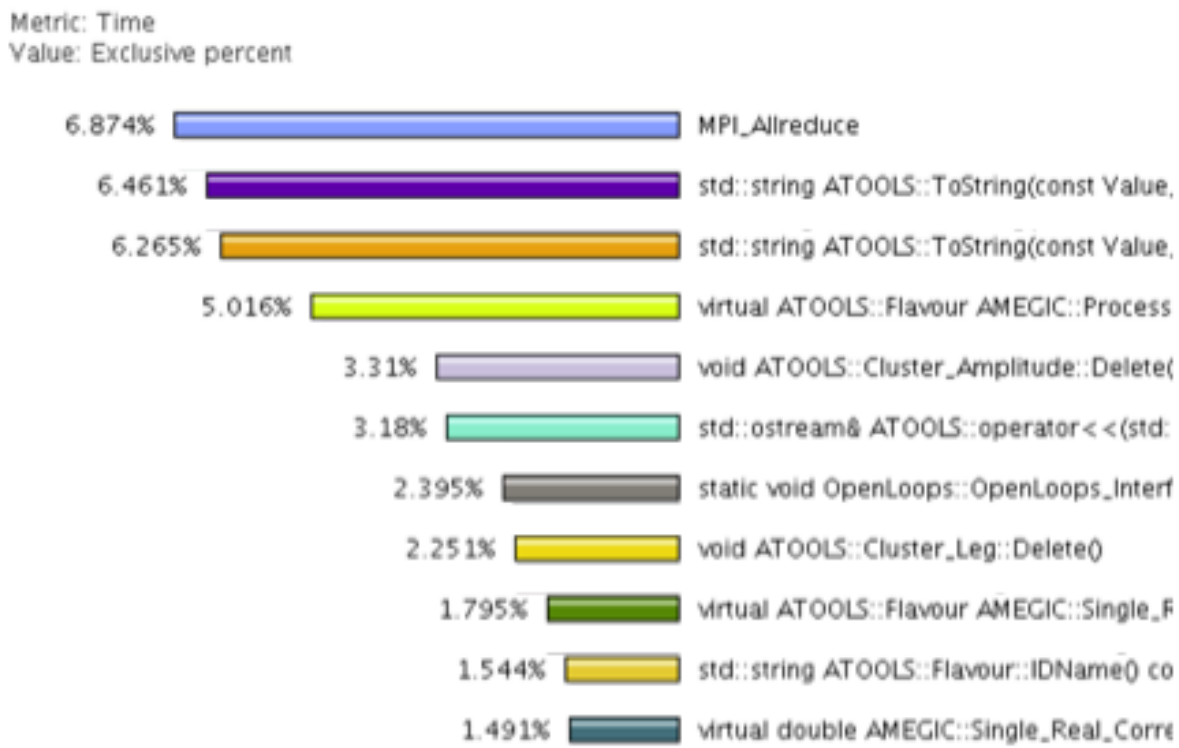
ME integration – scaling

Childers, Höche, LeCompte, Uram in prep.



ME integration – scaling

Benjamin, Childers, Höche, LeCompte, Uram in prep.



- most time spent in MPI communication
- also programming improvements mandatory

Asynchronous MPI

Problem: compute nodes sit idle during MPI comm time until they receive their answer (updated VEGAS grids)

- SHERPA is threaded, one thread handles MPI while the other runs the integration
- when the MPI thread has finished an `Allreduce` call, it interupts the calculator thread
- at this point, the calculator thread switches to the next process and continues, while the MPI thread calls an `Allreduce` on the process that has just been computed

This scheme is very efficient, except for the fact that we need two threads and therefore we may lose up to a factor of 2 globally in the runtime. In return, however, we will not waste any large amount of time wainting for an MPI answer that could be spent computing points.

Asynchronous MPI

Catch: integration results are not reproducible, because

- number of points for each optimisation step depends on MPI overhead
- updated VEGAS grids depend on the points used in optimisation (they are of course statistically compatible)

Still, on HPC machines this is a sensible route, as

- any calculation dominated by MPI communication the core now can do actual work (compute points) during the MPI wait loop
- eventual additional points generated may not enter the VEGAS optimisation, but will still improve the convergence of the integral (none of the effort is truly wasted)

Conclusions

- SHERPA's parallelisation options in the public versions are completely adaquate for usage on small clusters with with $\mathcal{O}$(few thousand) cores and 4 GB/core memory
- SHERPA's parallelisation options for large HPCs with in development, challenges are
  - traditional MPI no longer an option due to large MPI wait time
    $\rightarrow$ asynchronous MPI is a solution
  - limited memory per core
    $\rightarrow$ multithreading can help
  - modularity, demands a certain amount of framework overhead
    $\rightarrow$ allows for efficient physics development and maintenance

http://sherpa.hepforge.org

Thank you for your attention!