Making extreme computations possible with virtual machines

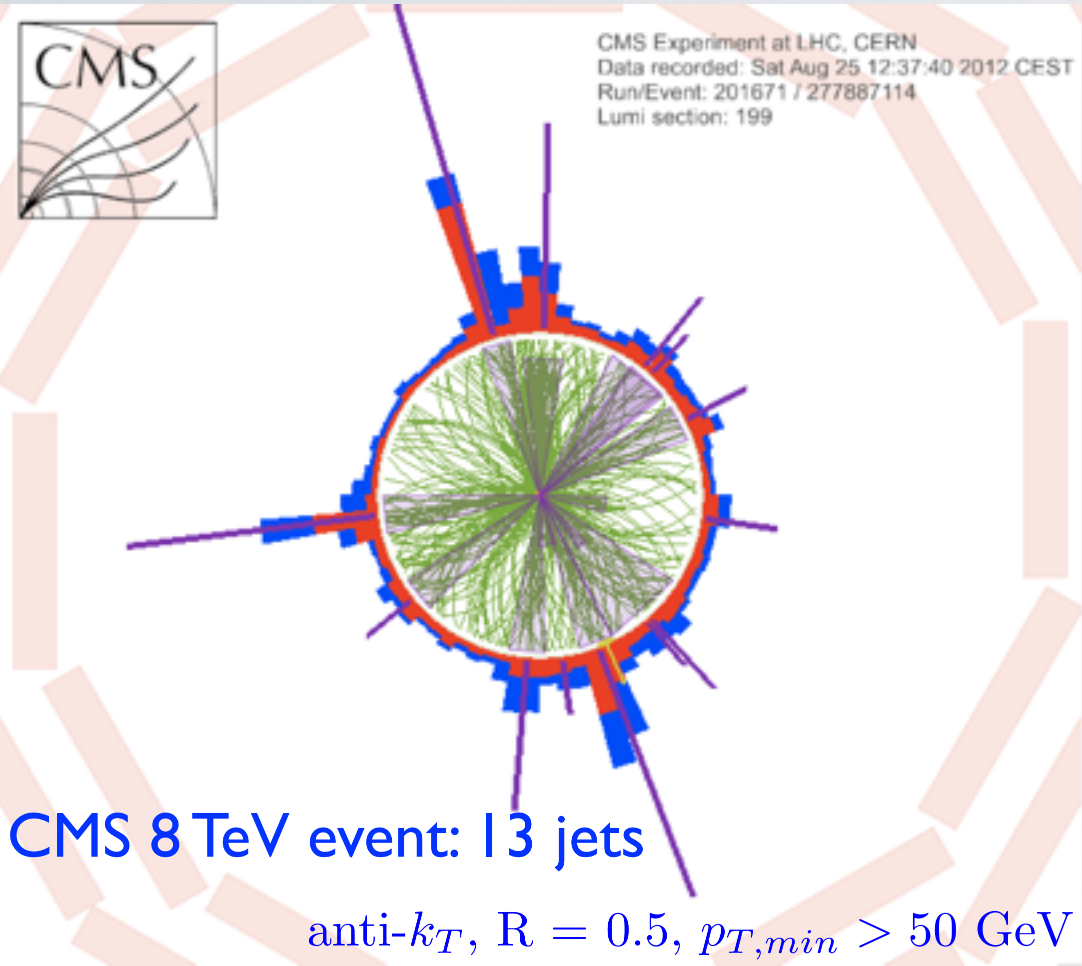
Jürgen R. Reuter, DESY

Chokoufé/Ohl/JRR, Comput.Phys.Commun. 196 (2015) 58-69

HELMHOLTZ
GEMEINSCHAFT

ACAT 2016
18-22 January 2016
UTFSM, Valparaíso (Chile)
Chile/Continental timezone

Motivation: why extreme computations?



CMS 8 TeV event: 13 jets

anti-k_T, R $= 0.5$, $p_{T,min} > 50$ GeV

- Modern collider experiments need matrix elements (MEs) of tremendous multiplicity
- Tree-level MEs needed in corrections to the parton shower (merging) and for *k* real emissions in N^{k}LO fixed order calculations
- Examples: access to top Yukawa at ILC/CLIC:

$$\begin{aligned} e^+e^- &\to \quad bbbb + 4j \\ e^+e^- &\to \quad bb\ell\nu_\ell + 6j \\ e^+e^- &\to \quad bb + 8j \end{aligned}$$

- Automated efficient ME generators provide every multiplicity (*in principle*) `Alpha` [Caravaglios/Moretti, 1995], `O'Mega` [Ohl/JRR, 2000], `Helac` [Papadopoulos, 2001], `Comix` [Gleisberg/Hoeche, 2008]

- Direct numerical implementations of recursions less flexible
- Traditional method: use meta programming to combine fast code & full flexibility
  - Algebraic expression from high-level language: `Form`, `Mathematica`, `OCaml`, `Python`, ...
  - Evaluation in numerical fast language: `C`, `Fortran`, ...
  - Examples: `O'Mega` [Ohl/JRR, 2000], `Madgraph` [Alwall et al., 2008], `FormCalc` [Hahn/Perez-Victoria, 1998], `GoSam` [van Deurzen et al., 2013], ...

What to do with complex processes?

- BUT: analytic expressions of complex (multi-jet) processes can reach GiB size,
- $gg \to 6g$ reaches size of 4 GB in `O'Mega` [Kilian/Ohl/JRR/Speckner, 1206.3700]
- No hope for higher multiplicities …
- Possible solution: Virtual Machine (VM) circumvents compilation of large code completely
- VM easy to implement and parallelize, similar performance than compiled code
- VM is no OS emulation in this context … … … [Chokoufé/Ohl/JRR, 1411.3834]

What is a Virtual Machine (VM)

☐ A Virtual Machine (VM) in our context is a compiled program (interpreter)

☐ The VM is able to read instructions from disk

☐ The VM performs arbitrary number of operations of a finite instruction set

☐ Instructions can be stored as byte code (encoded as numbers in ASCII file)

☐ VM can be regarded as machine with registers and instructions what to do on the registers

☐ Similar to a CPU, but on a higher level:

registers are e.g. arrays of momenta and wave functions

instructions are e.g. scalar products of momenta and wave functions

Contents of the byte code

- VM should be constructed dynamically (for user-defined processes)
- $\Longrightarrow$ provide header with number of objects to be allocated
- [optionally: version numbers to specify the physical model, comments on the creation of the byte code, tables with precomputed properties (information over helicity, color, flavor]
- Then: body of instructions (non-trivial information how to compute a process)
- First object of instruction is the opcode: specifies which operation is executed

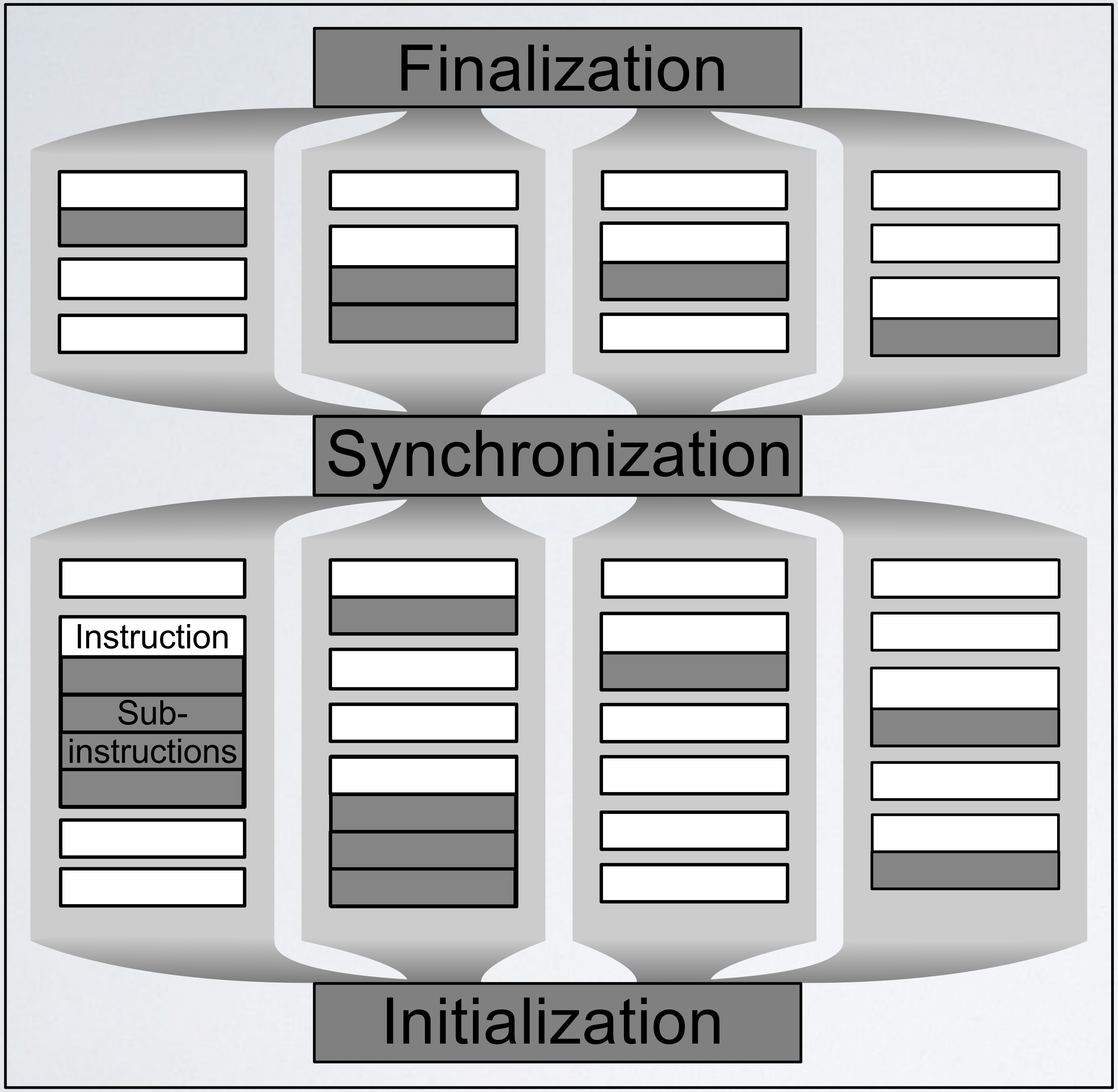
usually by addresses of registers, for example:

```
1 7 4 3    ⇔    momentum(7) = momentum(4) + momentum(3)
```

Interpreter (VM *per se*)

- Simple (compiled) program: reads first byte code into memory
- VM decode function loops over instructions with given input values
- Translation of byte code to machine code (very) fast compared to execution
  (execution consists of large number of complex scalar products)
- Adapting interpreter to a new process/matrix element requires
  - Specification of static information (spin tables, color tables etc.)
  - Writing the selection statements for the decode function
- VM is compiled once fast: handy for validation (checking many small processes), inevitable for (very) large processes

Parallelization in the Virtual Machine



- Group instructions into building blocks: minimize number of synchronization points

- Divide computation into levels

- All building blocks commute in every level

  (i.e. only one thread is writing to register per level)

Parallelization in High Energy Physics

- Usual assumption: trivial parallelizability of computations by computing multiple phase space points at once
  (phase space integration ≡ momenta, helicities, colors, flavors)

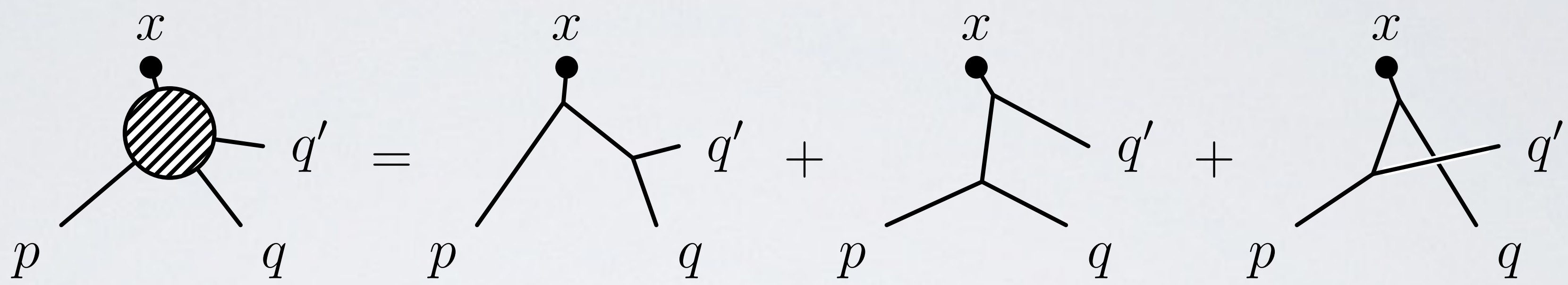
- Extreme computations: objects of single phase space point might already fill up your cache)

- Computing multiple points at once can induce traffic jam between RAM and CPU ⟹ might even be slower than single core performance

- VM is a straightforward implementation of the
  parallel computation of a single phase space point

The Optimizing Matrix Element Generator (`O'Mega`)

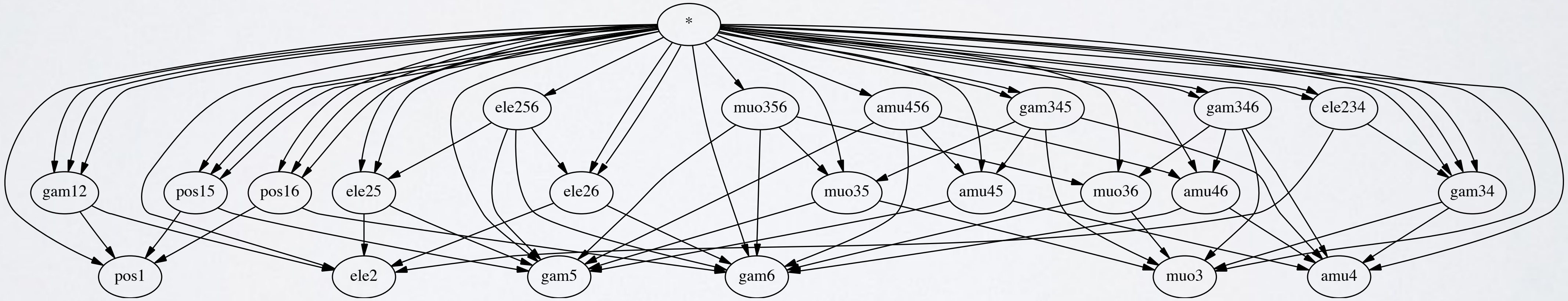
- `O'Mega` [Ohl, 2000; Moretti/Ohl/JRR, 2001; JRR, 2002] computes amplitudes with 1-particle off-shell wave functions (1POWs)



- Possible to construct set of all currents recursively (tree-/1-loop level)

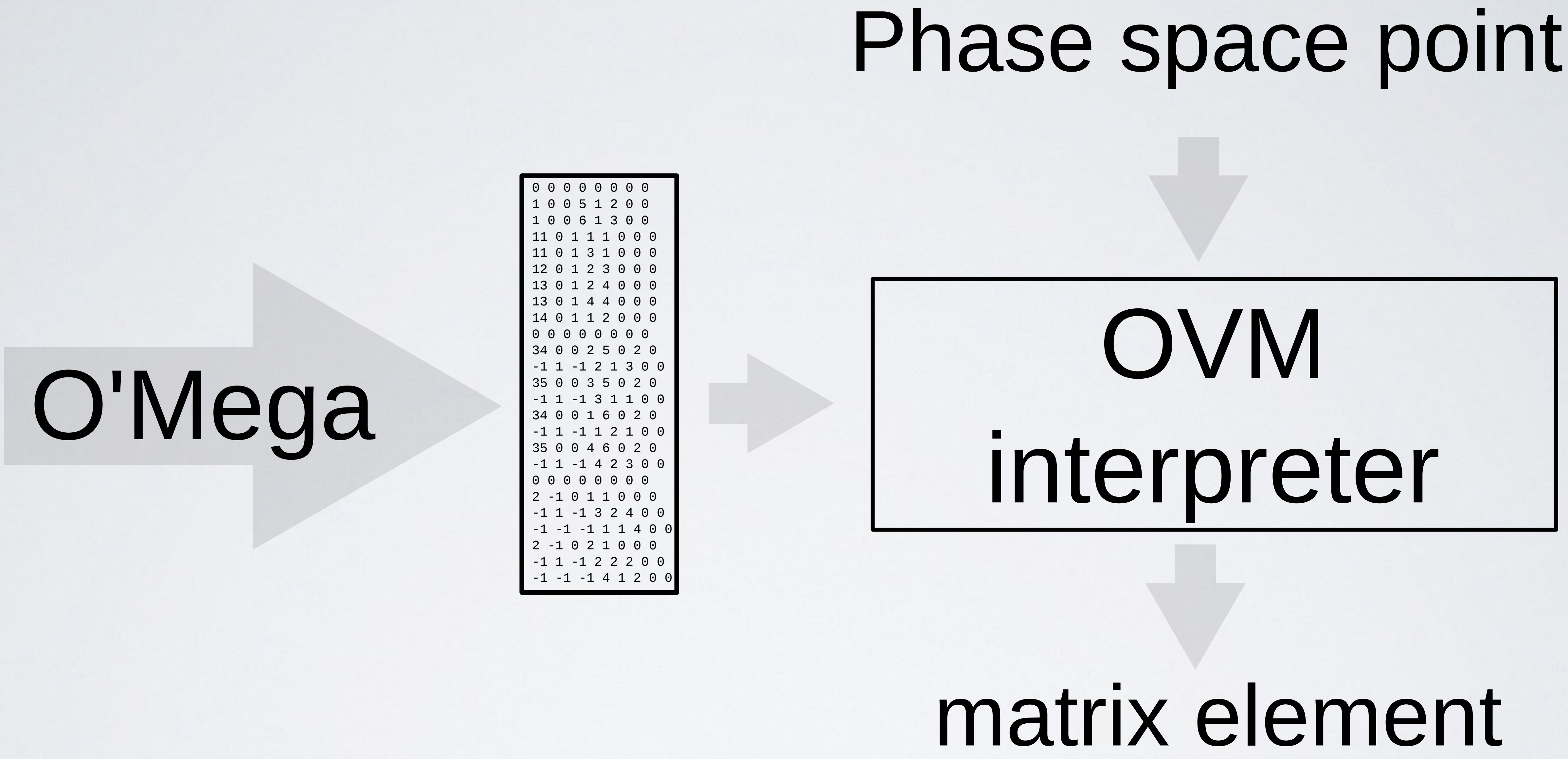
- Keystones *K* to replace sum over Feynman diagrams

$$\sum_{i=1}^{F(n)} D_i = \sum_{k,l,m=1}^{P(n)} K^{(3)}_{f_k f_l f_m}(p_k, p_l, p_m) W_{f_k}(p_k) W_{f_l}(p_l) W_{f_m}(p_m)$$



- Calculation forms Directed Acyclical Graphs (DAGs), optimized to consist only of the minimal number of connections by `O'Mega`

Layout of the general infrastructure



O'Mega

```
0 0 0 0 0 0 0 0
1 0 0 5 1 2 0 0
1 0 0 6 1 3 0 0
11 0 1 1 1 0 0 0
11 0 1 3 1 0 0 0
12 0 1 2 3 0 0 0
13 0 1 2 4 0 0 0
13 0 1 4 4 0 0 0
14 0 1 1 2 0 0 0
0 0 0 0 0 0 0 0
34 0 0 2 5 0 2 0
-1 1 -1 2 1 3 0 0
35 0 0 3 5 0 2 0
-1 1 -1 3 1 1 0 0
34 0 0 1 6 0 2 0
-1 1 -1 1 2 1 0 0
35 0 0 4 6 0 2 0
-1 1 -1 4 2 3 0 0
0 0 0 0 0 0 0 0
2 -1 0 1 1 0 0 0
-1 1 -1 3 2 4 0 0
-1 -1 -1 1 1 4 0 0
2 -1 0 2 1 0 0 0
-1 1 -1 2 2 2 0 0
-1 -1 -1 4 1 2 0 0
```

Phase space point

OVM
interpreter

matrix element

Byte code generation in `O'Mega/OCaml`

- Feynman rules form finite set of instructions
- Good candidates for translation into byte code
- Ordering of instructions needed
- `OCaml` compares abstract objects (wave functions, momenta, amplitudes)
- `Fortran` arrays ordered according to their index position
- Take set of objects: apply mapping to natural numbers using the given order

code	coupl	coeff	lhs	rhs$_1$	rhs$_2$	rhs$_3$	rhs$_4$
ADD_MOMENTA	0	0	p_lhs	p_rhs$_1$	p_rhs$_2$	p_rhs$_3$	0
LOAD_X	PDG	0	wf	outer_ind	0	0	amp
PROPAGATE_Y	PDG	width	wf	p	0	0	amp
FUSE_Z	coupl	coeff	lhs	rhs$_1$	rhs$_2$	rhs$_3$	rhs$_4$
CALC_BRAKET	sign	0	amp	sym	0	0	0

Performance of byte code generation

- ☑ Byte code is produced faster than compiled code
- ☑ Byte code uses less RAM than compiled code
- ☑ Byte code is a lot smaller than native (Fortran) source code

Performance of byte code generation

- ☑ Byte code is produced faster than compiled code
- ☑ Byte code uses less RAM than compiled code
- ☑ Byte code is a lot smaller than native (Fortran) source code

- ☑ Looking at (extreme) process $gg \to 6g$

  Memory requirements for code production reduced from 2.17 GiB to 1.34 GiB

  Code production time is reduced from 11 min 52 sec to 3 min 35 sec

Performance of byte code generation

- ☑ Byte code is produced faster than compiled code
- ☑ Byte code uses less RAM than compiled code
- ☑ Byte code is a lot smaller than native (Fortran) source code

- ☑ Looking at (extreme) process $gg \to 6g$

Memory requirements for code production reduced from 2.17 GiB to 1.34 GiB

Code production time is reduced from 11 min 52 sec to 3 min 35 sec

process	`BC` size	`Fortran` size	t_{compile}
$gg \to gggggg$	428 MB	4.0 GB	-
$gg \to ggggg$	9.4 MB	85 MB	483(18) s
$gg \to q\bar{q}q'\bar{q}'q''\bar{q}''g$	3.2 MB	27 MB	166(15) s
$e^+e^- \to e^+e^-e^+e^-e^+e^-e^+e^-e^+e^-$	0.7 MB	1.9 MB	32.46(13) s

Performance of byte code generation

- ☑ Byte code is produced faster than compiled code
- ☑ Byte code uses less RAM than compiled code
- ☑ Byte code is a lot smaller than native (Fortran) source code

- ☑ Looking at (extreme) process $gg \to 6g$

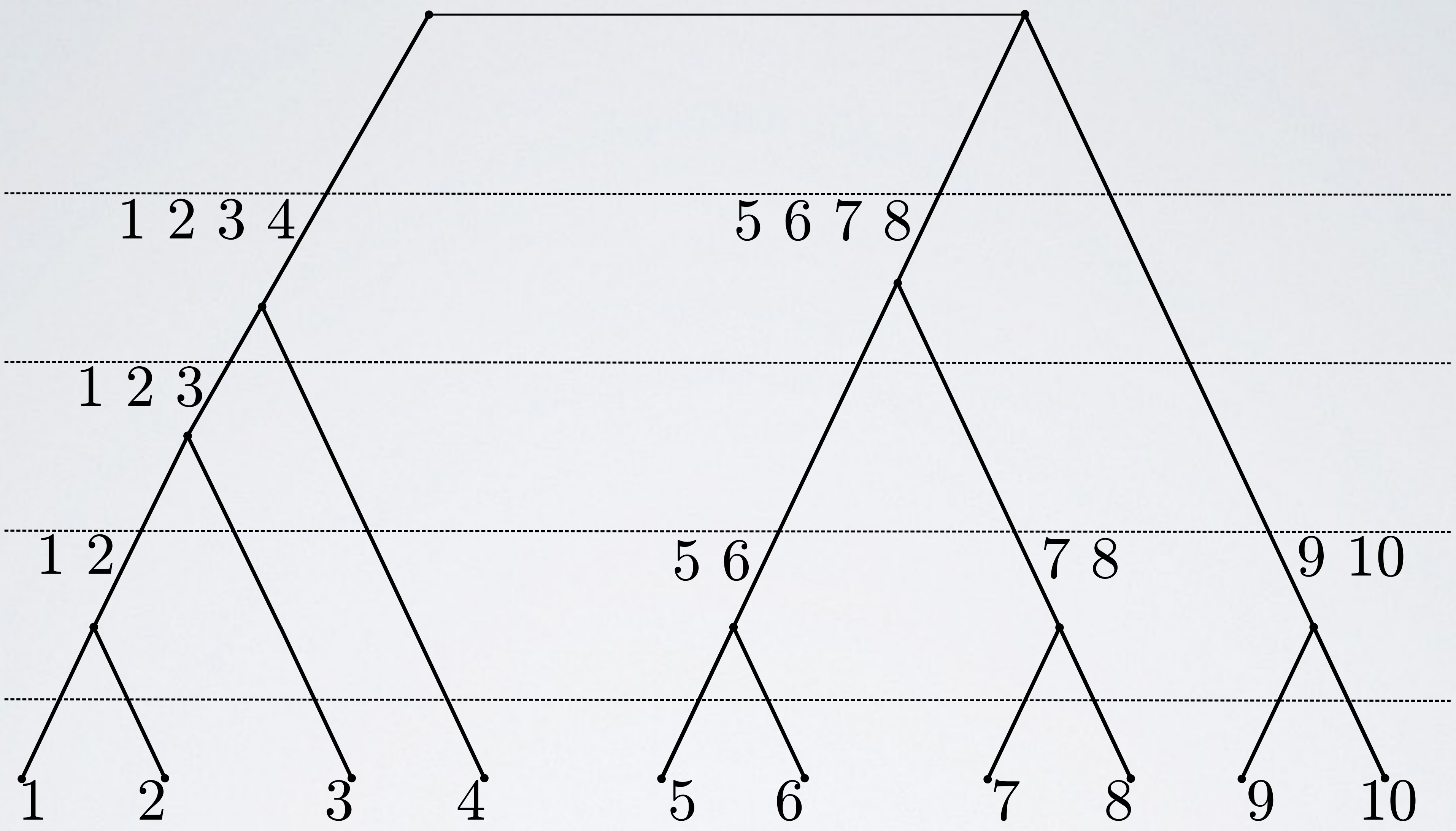
Memory requirements for code production reduced from 2.17 GiB to 1.34 GiB

Code production time is reduced from 11 min 52 sec to 3 min 35 sec

process	`BC` size	`Fortran` size	t_{compile}
$gg \to gggggg$	428 MB	4.0 GB	-
$gg \to ggggg$	9.4 MB	85 MB	483(18) s
$gg \to q\bar{q}q'\bar{q}'q''\bar{q}''g$	3.2 MB	27 MB	166(15) s
$e^+e^- \to e^+e^-e^+e^-e^+e^-e^+e^-e^+e^-$	0.7 MB	1.9 MB	32.46(13) s

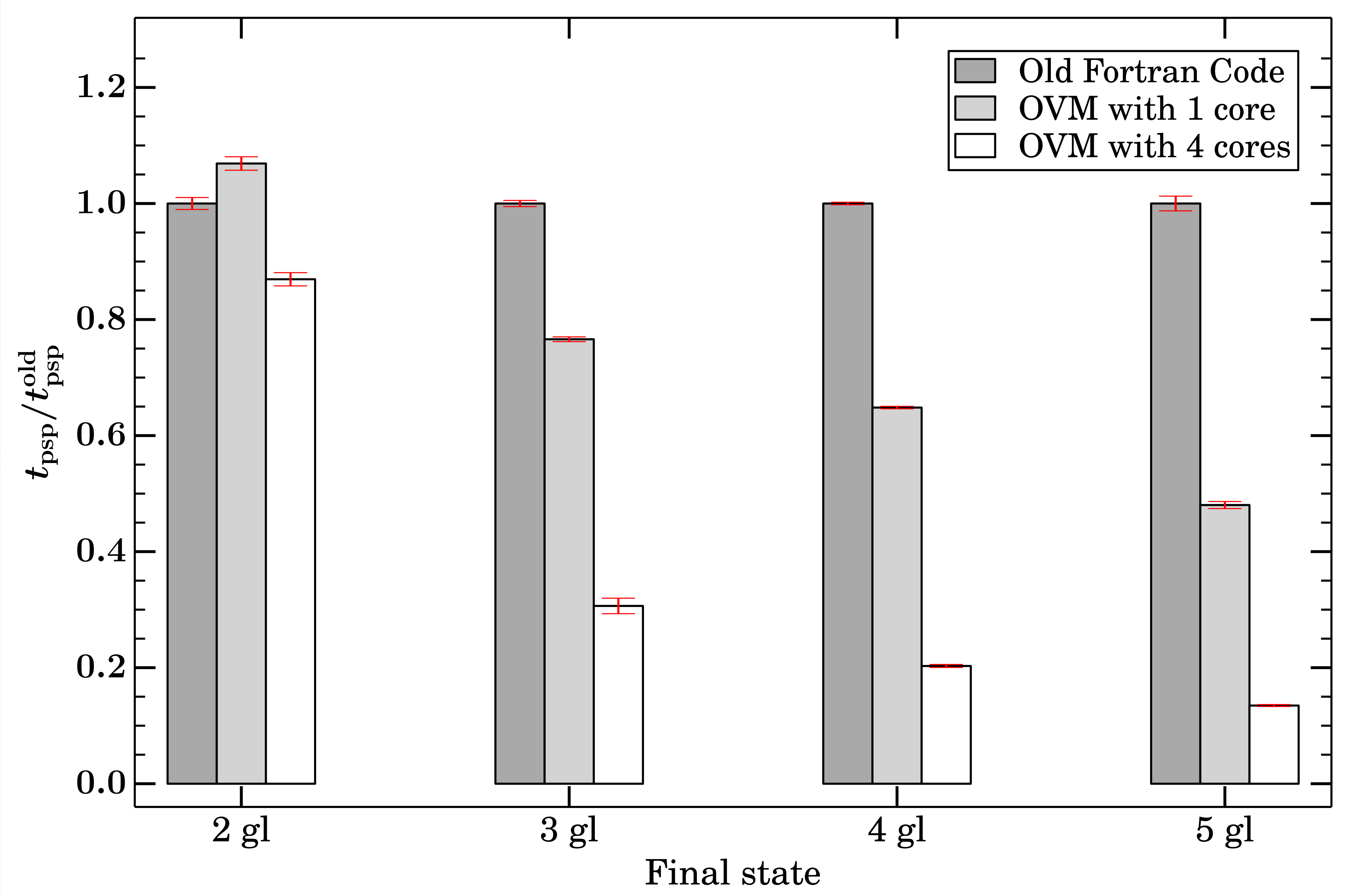
- ☑ No big changes for smaller processes

Parallelization in recursive computations



Identify a level of the parallelization by counting external momenta

Speed of VM matrix elements — First glance

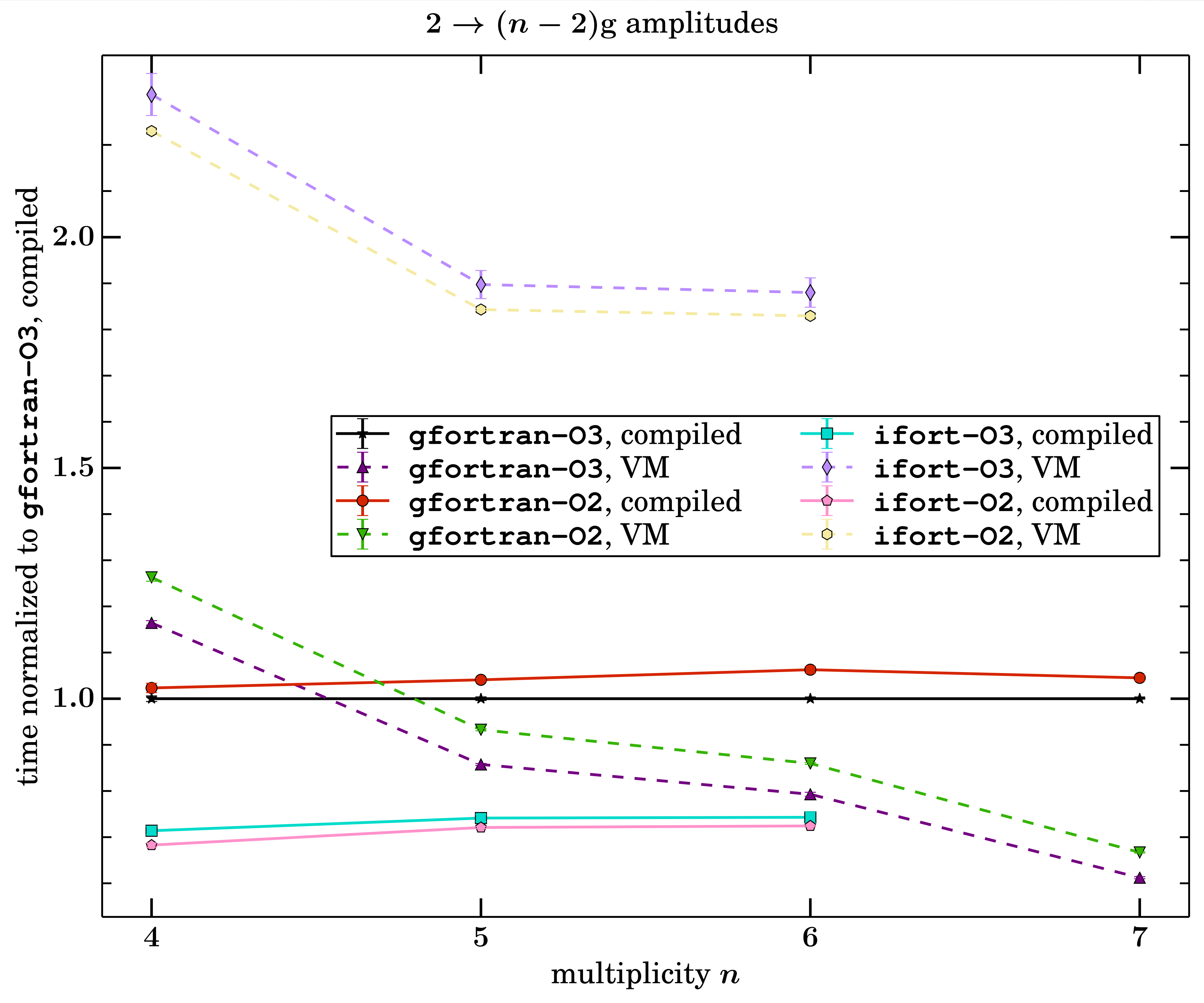


`[gfortran 4.7.4, Intel i7-2720QM @ 2.2 GHz]`

Speed of matrix elements — w/ two compilers

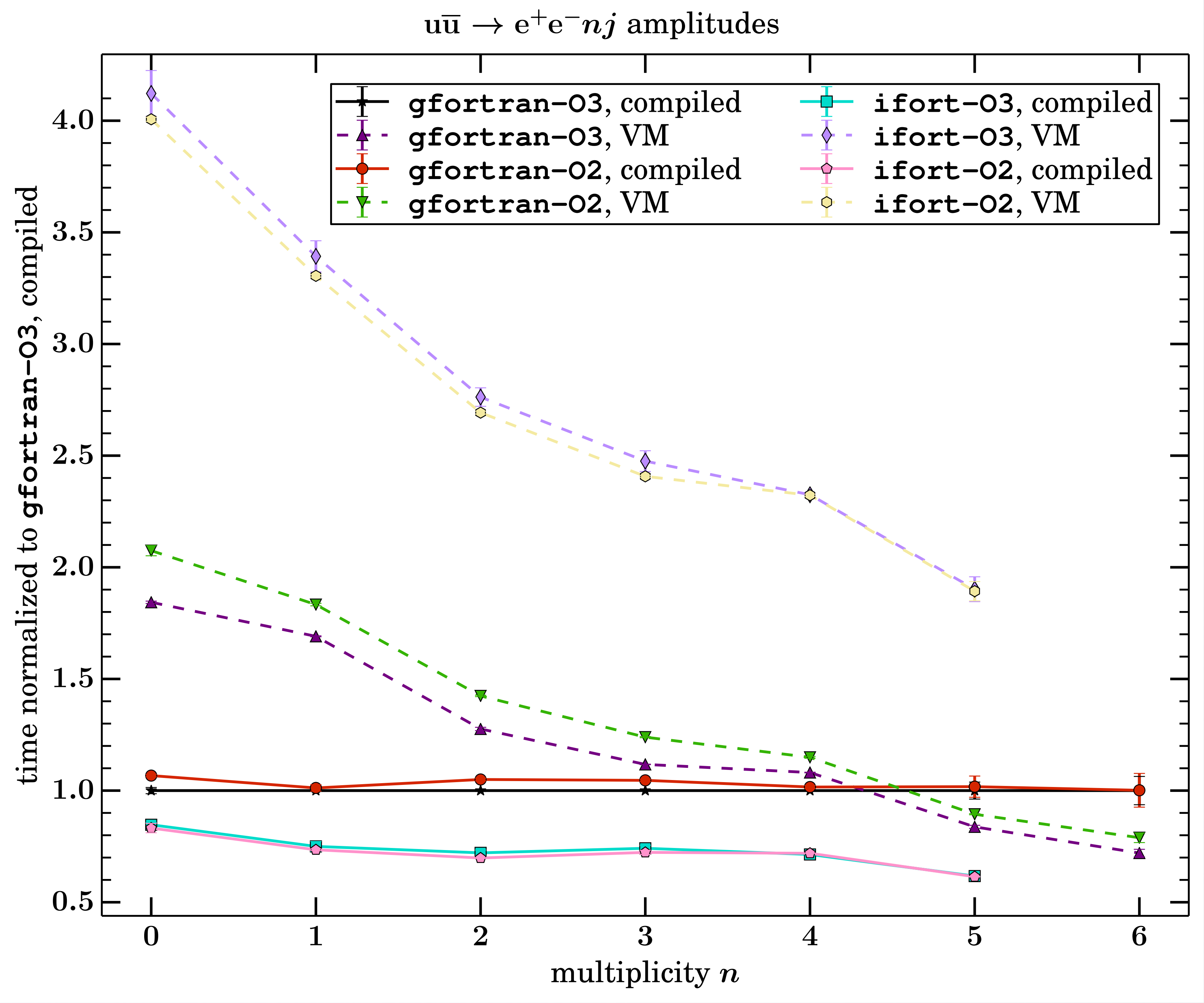
[`gfortran 4.7.4`/`ifort 14.0` on
`Intel Xeon E5-2440 @ 2.4 GHz`]

- Both VMs improve with increasing multiplicity
- ifort has large offset for the VM: needs profile-guided optimization to resolve
- `ifort v14.0` fails to compile 2 → 5 gluon amplitude
  (even with `-O0`)



Speed of matrix elements — explaining the Scaling

- Same scaling behavior
- Virtualisation costs constant
- VM does loop over levels and instructions in the levels
- Source code is like an unrolled version of this loop
- Double loop of VM has higher probability to keep decode function in instruction cache

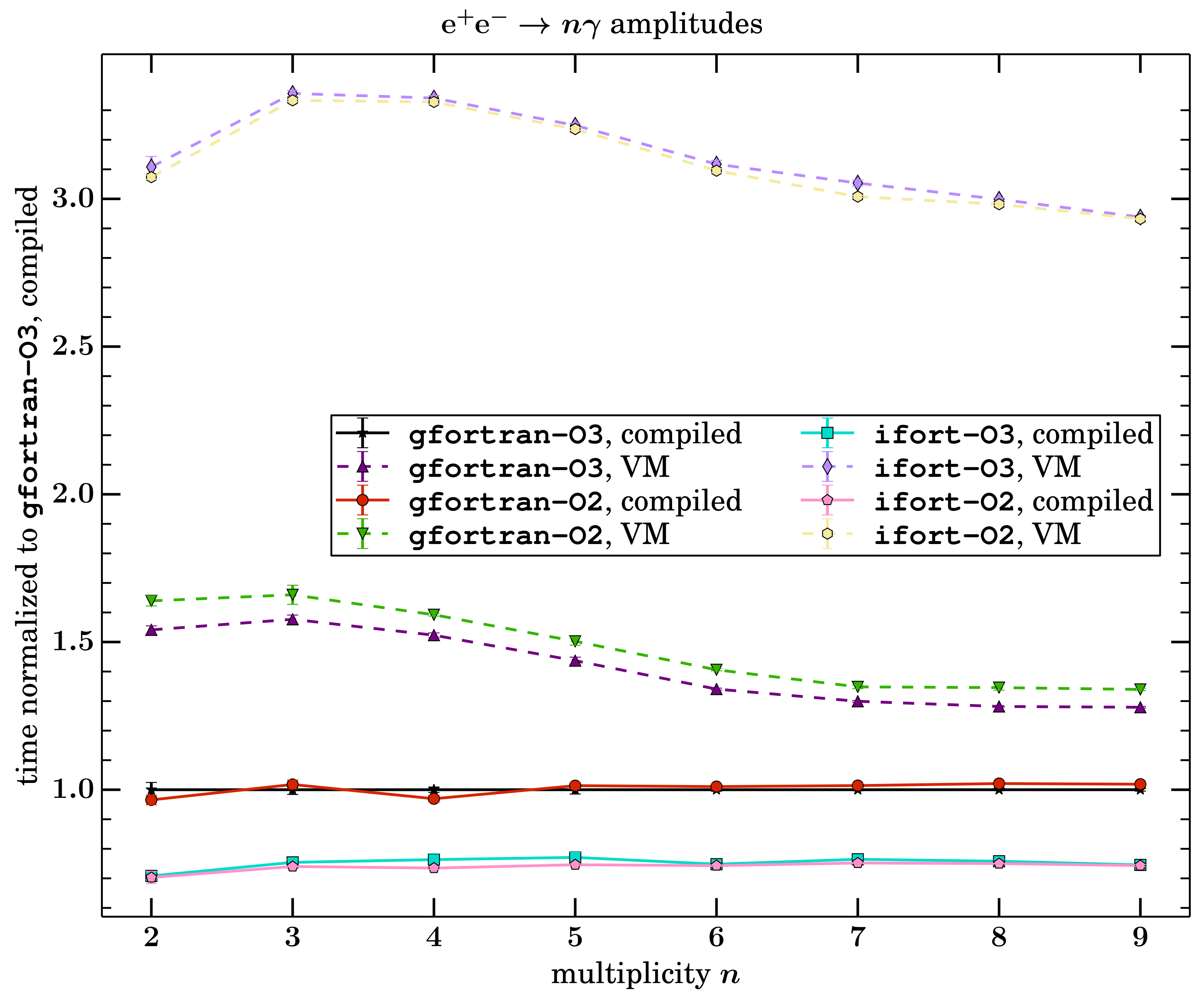


Speed of matrix elements

- For pure QED with increasing multiplicity: Improvements in VM smaller
- More wave functions per level: less to be done!
- Unrolled version can gain more from prefetching
- Be aware of the size:

  $e^+e^- \to 9\gamma$ 125 KiB

  $gg \to 4g$ 269 KiB



Quantitative Analysis of the Parallelization

- **Amdahl's Law** divides an algorithm into a parallelizable part *p* and strictly serial parts 1 - *p*

- **Amdahl's Law** determines possible speedup *s* for a computation with *n* cores

  Consider the time on 1 and *n* cores, resp.: $t^{(1)}, t^{(n)}$

Quantitative Analysis of the Parallelization

- **Amdahl's Law** divides an algorithm into a parallelizable part *p* and strictly serial parts 1 - *p*

- **Amdahl's Law** determines possible speedup s for a computation with *n* cores

Consider the time on 1 and *n* cores, resp.: $t^{(1)}, t^{(n)}$

$$\boxed{s \equiv \frac{t^{(1)}}{t^{(n)}} = \frac{1}{(1-p)+\frac{p}{n}} \leq \frac{1}{1-p}}$$

Quantitative Analysis of the Parallelization

- **Amdahl's Law** divides an algorithm into a parallelizable part *p* and strictly serial parts 1 - *p*

- **Amdahl's Law** determines possible speedup s for a computation with *n* cores

  Consider the time on 1 and *n* cores, resp.: $t^{(1)}, t^{(n)}$

$$\boxed{s \equiv \frac{t^{(1)}}{t^{(n)}} = \frac{1}{(1-p)+\frac{p}{n}} \leq \frac{1}{1-p}}$$

- Idealized case: communication costs between *n* cores have been neglected in the denominator $\propto c_1 \cdot n \left[+c_2 \cdot n^2\right]$

Quantitative Analysis of the Parallelization

- **Amdahl's Law** divides an algorithm into a parallelizable part p and strictly serial parts $1 - p$

- **Amdahl's Law** determines possible speedup s for a computation with *n* cores

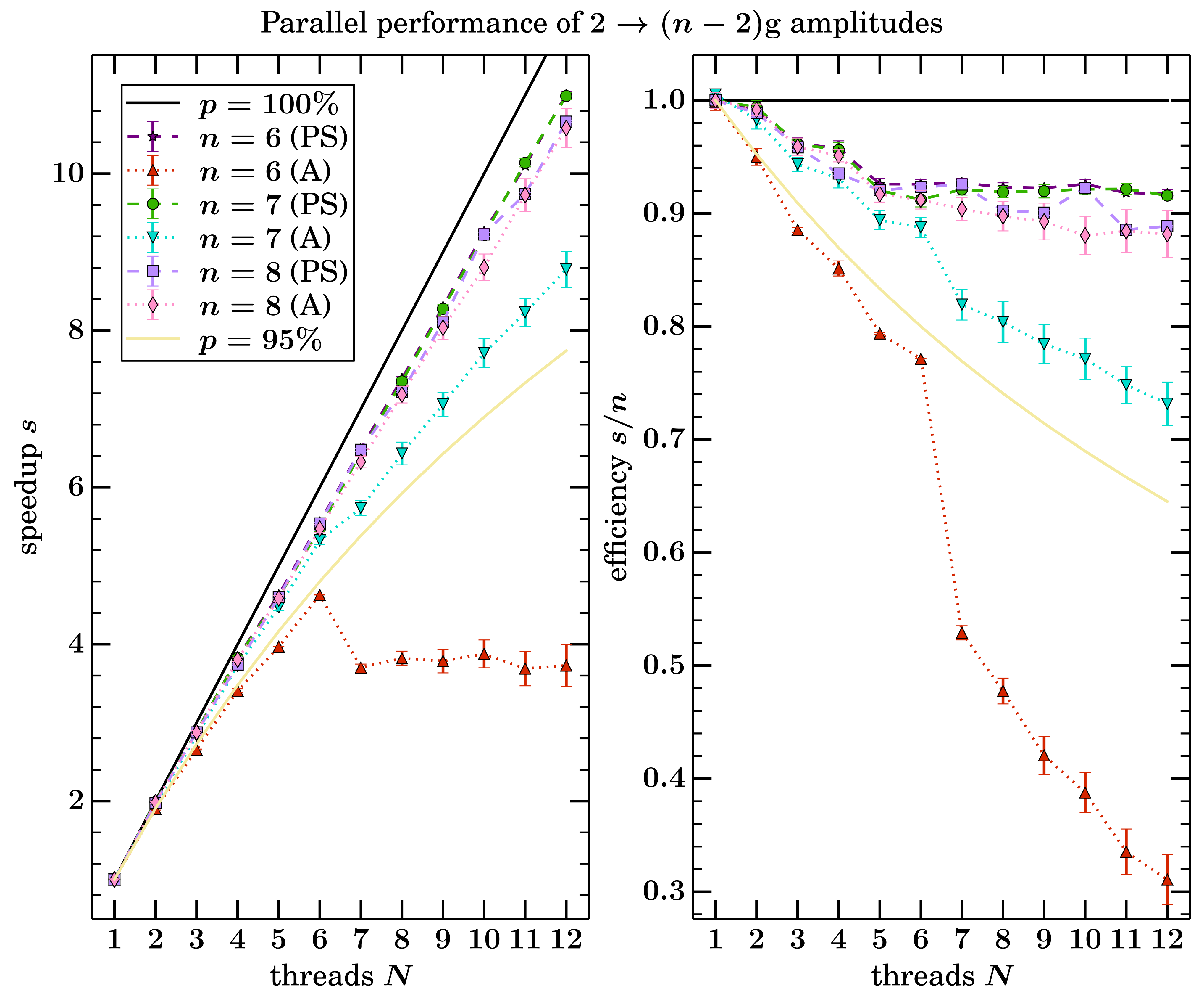
  Consider the time on 1 and *n* cores, resp.: $t^{(1)}, t^{(n)}$

$$\boxed{s \equiv \frac{t^{(1)}}{t^{(n)}} = \frac{1}{(1-p) + \frac{p}{n}} \leq \frac{1}{1-p}}$$

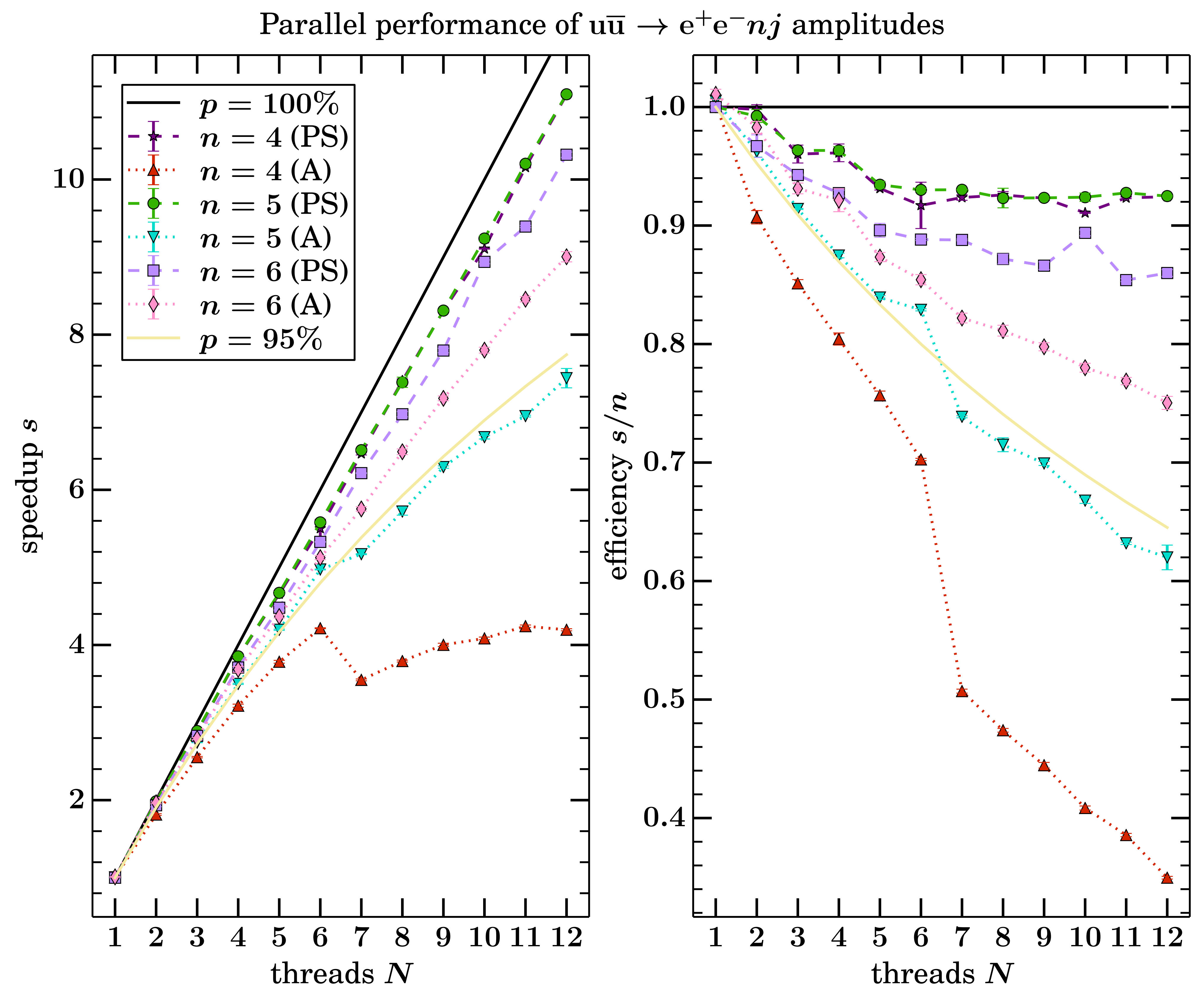
- Idealized case: communication costs between *n* cores have been neglected in the denominator $\propto c_1 \cdot n \left[+ c_2 \cdot n^2 \right]$

- Compare parallelization of the amplitude (A) with parallel computation of single complete phase space points (P)

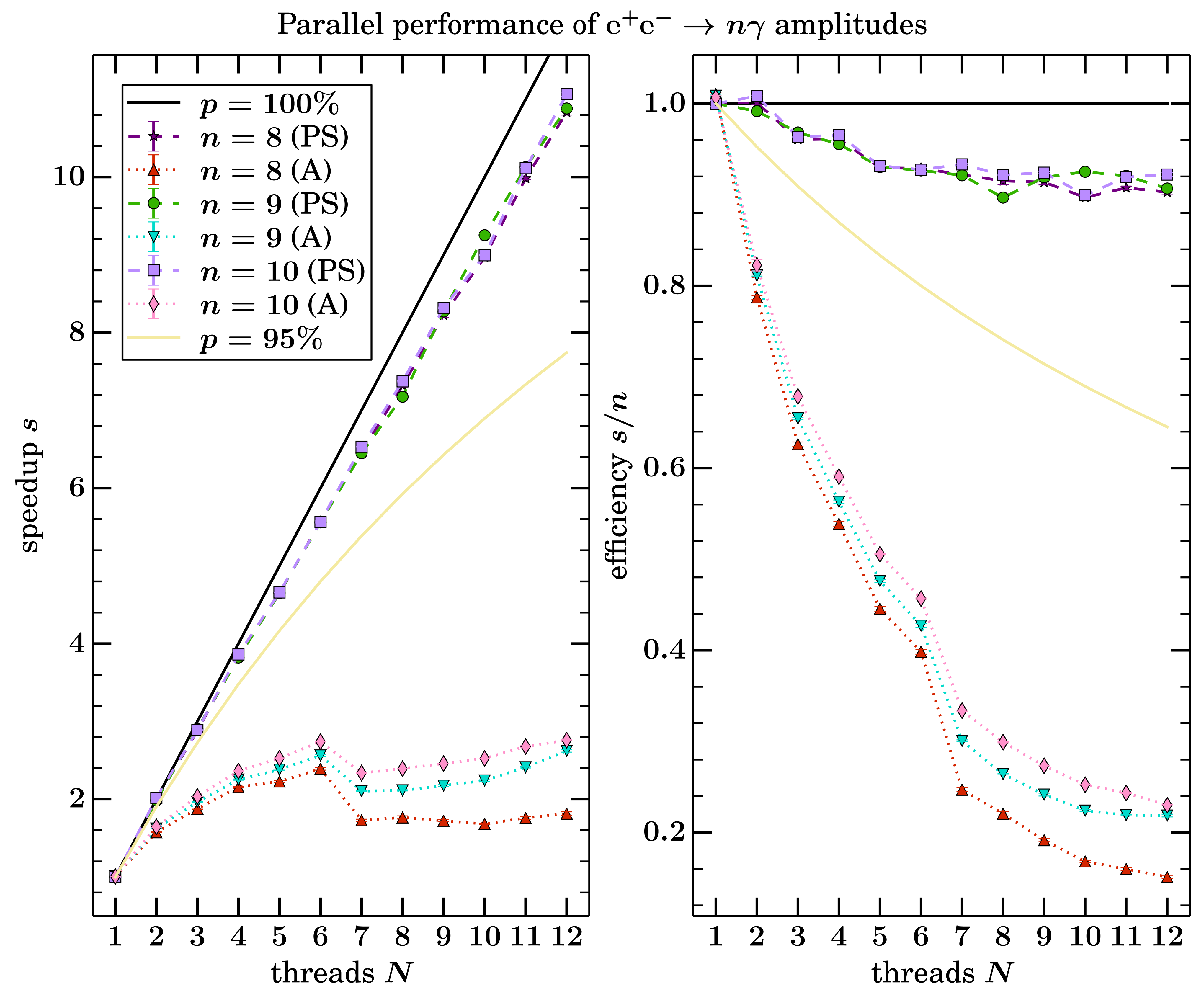
Quantitative Analysis of the Parallelization



Quantitative Analysis of the Parallelization



Quantitative Analysis of the Parallelization



Implementation / Usage in WHIZARD

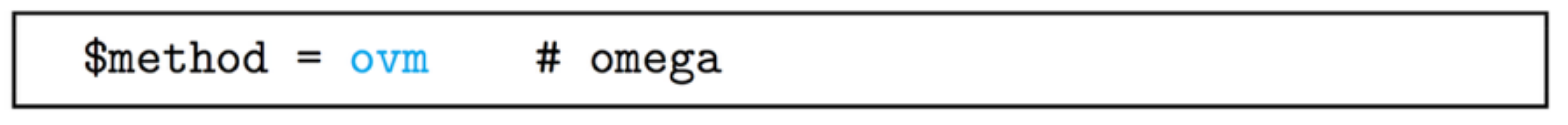
WHIZARD v2.2.8 (22.11.2015) *http://whizard.hepforge.org* <*whizard@desy.de*>

WHIZARD Team: *Wolfgang Kilian, Thorsten Ohl, JRR, Simon Braß/Bijan Chokoufé/Marco Sekulla/Soyoung Shim/Florian Staub/Christian Weiss/Zhijie Zhao* + *2 Master*

EPJ C71 (2011) 1742

- Universal event generator for lepton and hadron colliders
- Modular package:
  - Phase space parameterization (resonances, collinear emission, Coulomb etc.)
  - O'Mega optimized matrix element generator (recursiveness via Directed Acyclical Graphs)

    Compiled matrix element code & O'Mega Virtual Machine (OVM)
  - Color flow formalism Stelzer/Willenbrock, 2003; Kilian/Ohl/JRR/Speckner, 2011
  - VAMP: adaptive multi-channel Monte Carlo integrator
  - CIRCE1/2: generator/simulation tool for lepton collider beam spectra
  - Lepton beam ISR Kuraev/Fadin, 2003; Skrzypek/Jadach, 1991



```
$method = ovm     # omega
```

Available for the following models:

`SM, SM_CKM, SM_Higgs, Zprime, QCD, QED, 2HDM, 2HDM_CKM, HSExt`

Application on GPUs ?

- GPUs allow highly efficient 4-vector calculations (speed-ups of 50-100 reported)
- GPUs suffer from problem of finite (small) size of its kernel
- VM could be the perfect tool for such computations
- Existing studies: performance degrades with growing multiplicity ⟹ code has to be split into smaller programs [Hagiwara/Kanzaki/Li/Okamura/Stelzer, 1305.0708]
- Decode function of VM remains sufficiently small
- Possible remaining obstacle: Efficiency of memory management
- Reduce communication costs: transfer instructions and VM to GPU only once
- Strategy:

Application on GPUs ?

- GPUs allow highly efficient 4-vector calculations (speed-ups of 50-100 reported)
- GPUs suffer from problem of finite (small) size of its kernel
- VM could be the perfect tool for such computations
- Existing studies: performance degrades with growing multiplicity $\Longrightarrow$ code has to be split into smaller programs [Hagiwara/Kanzaki/Li/Okamura/Stelzer, 1305.0708]
- Decode function of VM remains sufficiently small
- Possible remaining obstacle: Efficiency of memory management
- Reduce communication costs: transfer instructions and VM to GPU only once
- Strategy:
  - Send only quantum numbers and momenta to GPU
  - Receive the amplitudes as complex numbers from the GPU
  - Phase space integration / reweighting etc. happens on the CPU

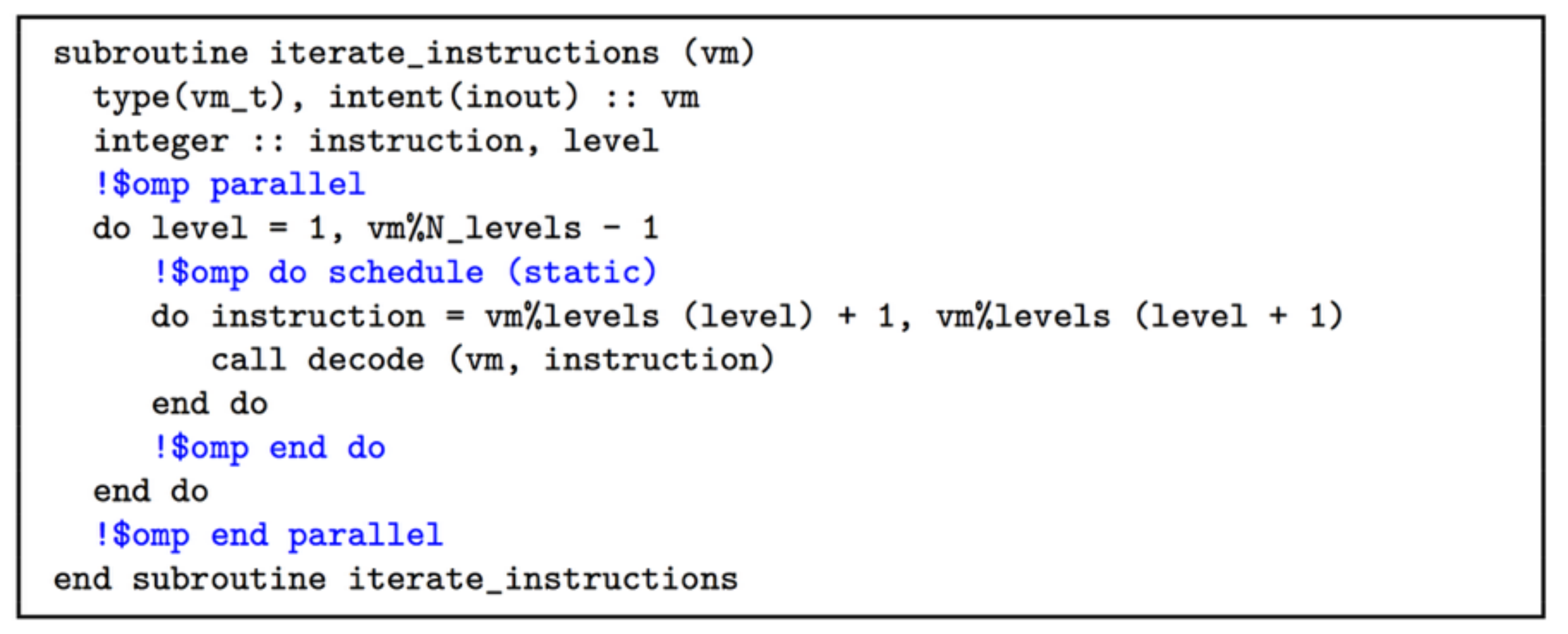
Conclusions & Outlook

- Virtual Machines allow to compute directly
- No compile time needed (hours/days for complicated processes)
- Implementation for High Energy Physics: O'Mega Virtual Machine

  [Chokoufé/Ohl/JRR, Comput.Phys.Commun. 196 (2015) 58-69]
- OVM included in event generator `WHIZARD 2.2`
- (Very) complicated processes benefit from/need parallelization of single phase space points   straightforward in the VM
- Execution times same order as compiled code (sometimes even faster)
- Might allow to run on graphic cards: ME [OVM] on GPU, MC core on CPU
- Idea very general: summation, algebraic operations, etc.

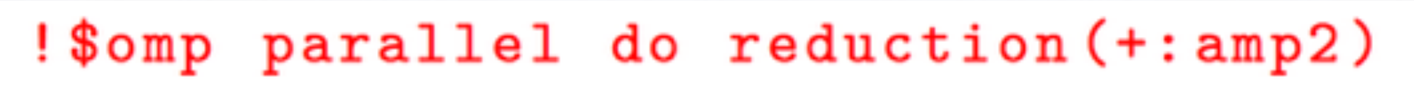
BACKUP SLIDES

Implementation of (OMP) parallelisation



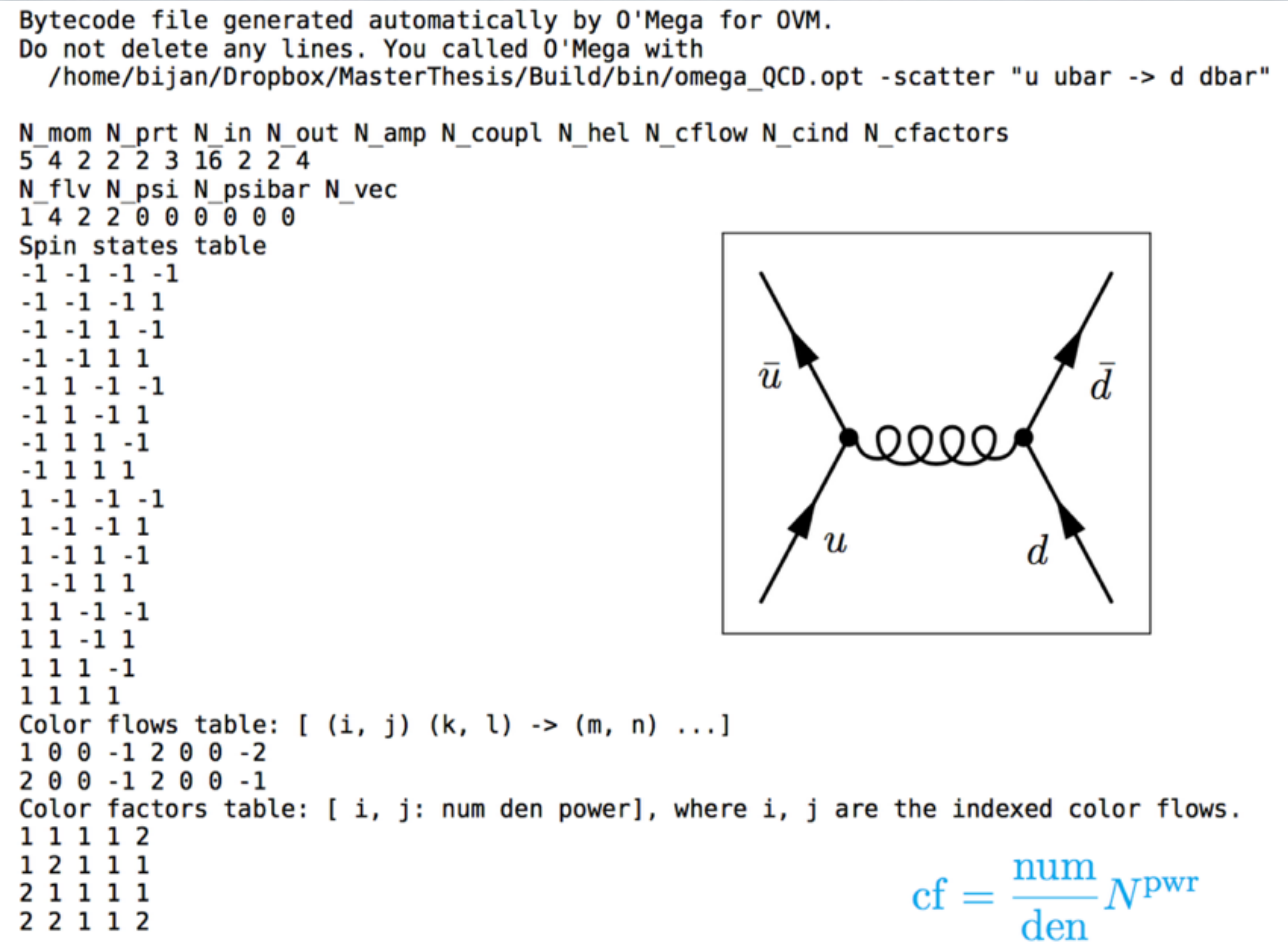
```
subroutine iterate_instructions (vm)
  type(vm_t), intent(inout) :: vm
  integer :: instruction, level
  !$omp parallel
  do level = 1, vm%N_levels - 1
     !$omp do schedule (static)
     do instruction = vm%levels (level) + 1, vm%levels (level + 1)
        call decode (vm, instruction)
     end do
     !$omp end do
  end do
  !$omp end parallel
end subroutine iterate_instructions
```

Also the color sum has to be parallelized, too:



```
!$omp parallel do reduction(+:amp2)
```

Byte code in detail

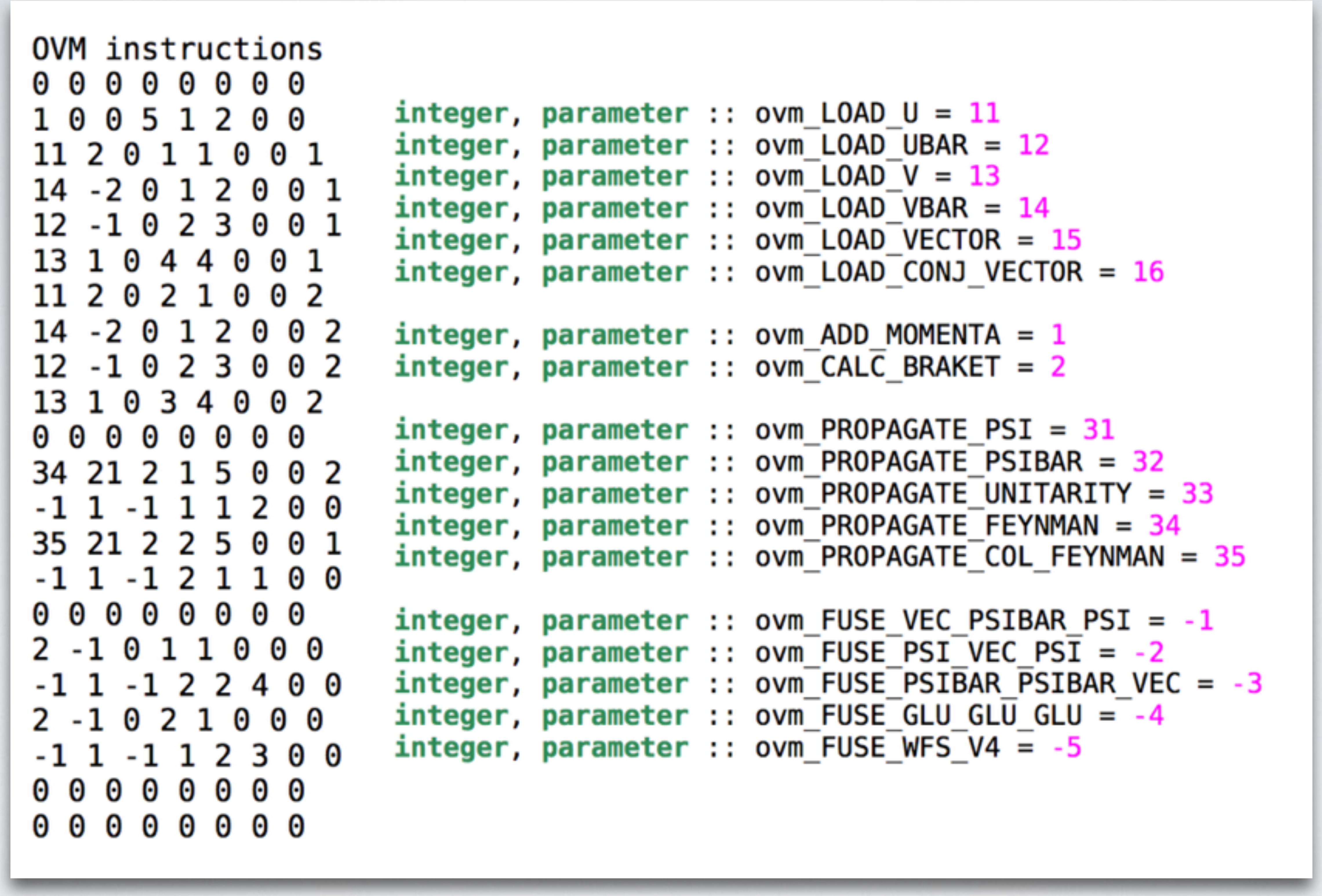


```
Bytecode file generated automatically by O'Mega for OVM.
Do not delete any lines. You called O'Mega with
  /home/bijan/Dropbox/MasterThesis/Build/bin/omega_QCD.opt -scatter "u ubar -> d dbar"

N_mom N_prt N_in N_out N_amp N_coupl N_hel N_cflow N_cind N_cfactors
5 4 2 2 2 3 16 2 2 4
N_flv N_psi N_psibar N_vec
1 4 2 2 0 0 0 0 0 0
Spin states table
-1 -1 -1 -1
-1 -1 -1 1
-1 -1 1 -1
-1 -1 1 1
-1 1 -1 -1
-1 1 -1 1
-1 1 1 -1
-1 1 1 1
1 -1 -1 -1
1 -1 -1 1
1 -1 1 -1
1 -1 1 1
1 1 -1 -1
1 1 -1 1
1 1 1 -1
1 1 1 1
Color flows table: [ (i, j) (k, l) -> (m, n) ...]
1 0 0 -1 2 0 0 -2
2 0 0 -1 2 0 0 -1
Color factors table: [ i, j: num den power], where i, j are the indexed color flows.
1 1 1 1 2
1 2 1 1 1
2 1 1 1 1
2 2 1 1 2
```

$$\mathrm{cf} = \frac{\mathrm{num}}{\mathrm{den}} N^{\mathrm{pwr}}$$

Byte code in detail



```
OVM instructions
0 0 0 0 0 0 0 0
1 0 0 5 1 2 0 0
11 2 0 1 1 0 0 1
14 -2 0 1 2 0 0 1
12 -1 0 2 3 0 0 1
13 1 0 4 4 0 0 1
11 2 0 2 1 0 0 2
14 -2 0 1 2 0 0 2
12 -1 0 2 3 0 0 2
13 1 0 3 4 0 0 2
0 0 0 0 0 0 0 0
34 21 2 1 5 0 0 2
-1 1 -1 1 1 2 0 0
35 21 2 2 5 0 0 1
-1 1 -1 2 1 1 0 0
0 0 0 0 0 0 0 0
2 -1 0 1 1 0 0 0
-1 1 -1 2 2 4 0 0
2 -1 0 2 1 0 0 0
-1 1 -1 1 2 3 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

```
integer, parameter :: ovm_LOAD_U = 11
integer, parameter :: ovm_LOAD_UBAR = 12
integer, parameter :: ovm_LOAD_V = 13
integer, parameter :: ovm_LOAD_VBAR = 14
integer, parameter :: ovm_LOAD_VECTOR = 15
integer, parameter :: ovm_LOAD_CONJ_VECTOR = 16

integer, parameter :: ovm_ADD_MOMENTA = 1
integer, parameter :: ovm_CALC_BRAKET = 2

integer, parameter :: ovm_PROPAGATE_PSI = 31
integer, parameter :: ovm_PROPAGATE_PSIBAR = 32
integer, parameter :: ovm_PROPAGATE_UNITARITY = 33
integer, parameter :: ovm_PROPAGATE_FEYNMAN = 34
integer, parameter :: ovm_PROPAGATE_COL_FEYNMAN = 35

integer, parameter :: ovm_FUSE_VEC_PSIBAR_PSI = -1
integer, parameter :: ovm_FUSE_PSI_VEC_PSI = -2
integer, parameter :: ovm_FUSE_PSIBAR_PSIBAR_VEC = -3
integer, parameter :: ovm_FUSE_GLU_GLU_GLU = -4
integer, parameter :: ovm_FUSE_WFS_V4 = -5
```