Concurrent Cuba

Thomas Hahn

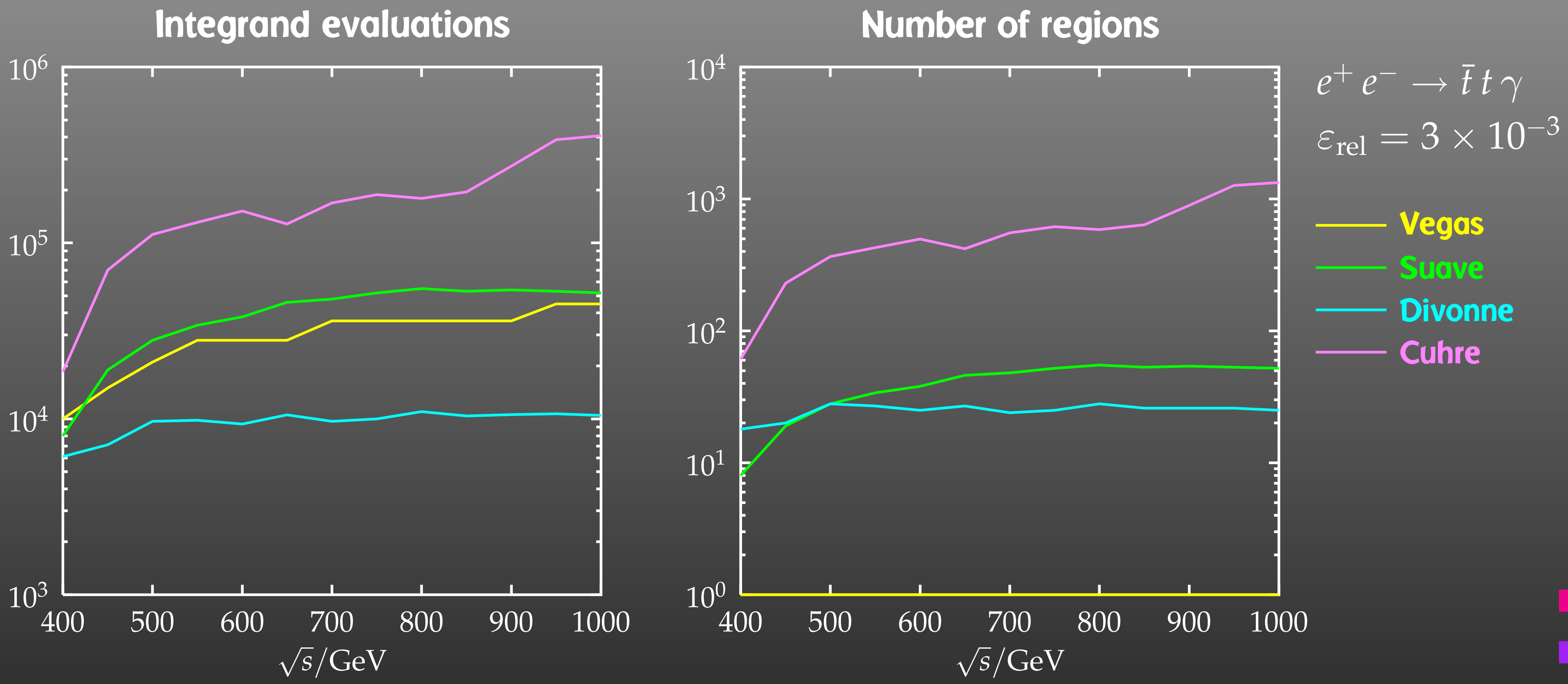
Max-Planck-Institut für Physik
München

Overview of the Cuba Routines

Routine	Basic method	Type	Variance reduction
Vegas	Sobol sample	quasi MC	importance sampling
	or MT sample	pseudo MC	
Suave	Sobol sample	quasi MC	globally adaptive subdivision
	or MT sample	pseudo MC	+ importance sampling
Divonne	Korobov sample	lattice MC	stratified sampling,
	or Sobol sample	quasi MC	aided by methods from
	or MT sample	pseudo MC	numerical optimization
	or cubature rules	deterministic	
Cuhre	cubature rules	deterministic	globally adaptive subdivision

- Very similar invocation (easily interchangeable)
- Fortran, C/C++, Mathematica interface provided
- Can integrate vector integrands

Cuba Comparison



‘Gauge’ integration problem first:

- Compute with all four routines.
- Check whether results are consistent.
- Select fastest algorithm.

Parallel Cuba

- In Mathematica:
  Parallelizes through Mathematica functions only, available since Cuba 2.
- In C/C++/Fortran:
  Parallel features available since Cuba 3.
- Extended in Cuba 4 for Accelerators (GPUs) and Vectorization.

Parallelization in Mathematica

- Mathematica interface works as follows:
  - Cuba sends coordinates to Mathematica.
  - Sampling is done in Mathematica.
  - Mathematica returns integrand values.

  Can sample any Mathematica function (e.g. `Zeta`).
- MathLink programs run independently, have 'external' (e.g. TCP) link to Mathematica Kernel (license issues).
- Cannot parallelize Kernel through OS functions thus. Parallelization only by Mathematica means.
- Sampling uses `MapSample`. By default `MapSample = Map`.
- To parallelize redefine `MapSample = ParallelMap`.
- Must use `DistributeDefinitions`, `ParallelNeeds` for required definitions, packages.

Parallelization Design Considerations

No additional software shall be needed.

- OS functions only.
- No parallelization across the network (e.g. via MPI).
- Uses internal cores 'only', thus e.g. 4 or 8.
- Speed-ups not expected to be linear anyway.
- More cores not necessarily useful.

Shall work for any integrand function.

- Requires user's understanding of issues (e.g. global variables, common blocks, I/O buffers).
- Re-coding effort for old code.
- Reentrancy cannot be fully controlled e.g. in Fortran.

Parallelization Design Considerations

Parallelization should work ‘automatically.’

- No system knowledge required.
- No re-compile necessary.
- Auto-detect # of cores + load at run-time.
- User control through environment variable `CUBACORES` or API calls.
- Auto-parallelization only acceptable if speed-ups ‘reasonable.’

Shall be available on all platforms.

- Native Windows has no `fork` function.
- Cygwin API emulates `fork` but quite slow.
- `fork` is moderately ‘expensive’ even on Linux/MacOS.
- Keep `fork` calls minimal: fork only at entry into Cuba routine.

Parallelization Design Considerations

Usual issues with parallel sample generation.

- How to independently seed parallel random-number generators?
- Best to generate samples on master only, distribute to workers.
- 1 Master, N workers on N-core system.

fork vs. pthread_create

- `pthread_create` creates additional thread in same memory space.
- `fork` creates completely independent process.
- On Linux: pages not actually duplicated until written on (‘copy-on-write’), thus no large penalty.
- No `fork` on native Windows (must use Cygwin).

Must use `fork` for non-reentrant integrands.

Master-Worker Communication

Possible communication channels:

- file read/write,
- pipe read/write,
- socket read/write,
- shared memory (IPC).

I/O creates obvious scheduling point for kernel.
Need semaphore or similar if using shared memory only.

Used in Cuba:

- (if available:) shared memory for samples,
- socketpair read/write for control information.

‘Simple’ Implementation

All Cuba routines:

- **Main sampling routine `DoSample` already abstracted in Cuba 1 since C/C++ and Mathematica implementations very different.**
- `DoSample` straightforward to parallelize on N cores:

Serial	sample n points
Parallel	send n/N points to core 1
	$\ldots$
	send n/N points to core N

- Fill fewer cores if not enough samples.

Implementation for Divonne

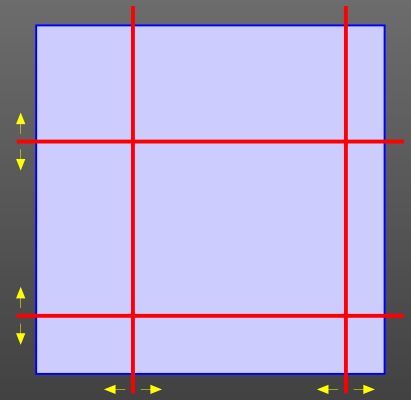
Divonne:

- Parallelizing `DoSample` alone not satisfactory. Speed-ups generally $\lesssim$ 1.5.
- Needs special treatment.
- Partitioning Phase significant.
- Partitioning originally recursive, had to 'un-recurse' algorithm first, mainly by better bookkeeping of regions.
- Each core receives entire region to subdivide, not just list of points.
- Efficiently distributes min/max search where only one point at a time is sampled.

Divonne Algorithm

- **Phase 1 – Partitioning**
  - For each subregion, 'actively' determine $\sup f$ and $\inf f$ using methods from numerical optimization.
  - Move 'dividers' around until all subregions have approximately equal spread, defined as

$$\mathrm{Spread}(r) = \frac{1}{2}\,\mathrm{Vol}(r)\left(\sup_{\vec{x}\in r} f(\vec{x}) - \inf_{\vec{x}\in r} f(\vec{x})\right).$$



- **Phase 2 – Sampling**
  Sample the subregions independently with the same number of points each. The latter is extrapolated from the results of Phase 1.
- **Phase 3 – Refinement**
  Further subdivide or sample again if results from Phase 1 and 2 do not agree within their error.

Accelerators and Cores

Based on the strategy used to distribute samples, Cuba distinguishes two kinds of workers:

- Accelerators (GPU).
- Cores (CPU).

Can have both kinds in same Cuba call.

Integrand can tell which it is running on by ‘core’ argument:

```
typedef int (*integrand_t)(
  const int *ndim, const double x[],
  const int *ncomp, double f[], void *userdata,
  const int *nvec, const int *core, ...);
```

$$
\begin{aligned}
*\texttt{core} &< 0 && \rightarrow \text{Accelerator,}\\
&\geqslant 0 && \rightarrow \text{Core,}\\
&= 32768 && \rightarrow \text{Master.}
\end{aligned}
$$

Accelerators Distribution Strategy

- Assumes device so highly parallel that sampling time is independent of number of points, up to hardware number of threads p_{accel}.
- Cuba sends exactly p_{accel} points to each core – never more, less only for the last batch.
- Example: Sampling 2400 points on 3 accelerators with $p_{\mathrm{accel}} = 1000$ gives 3 batches 1000/1000/400.
- Cuba does not actually send anything to a GPU or Accelerator. Can only be done by integrand function.

Cores Distribution Strategy

- All available cores are used.
- Points are distributed evenly.
- Example: Sampling 2400 points on 3 cores with p_{cores} = 1000 gives 3 batches 800/800/800.
- Each core receives $\geqslant$ 10 points, or fewer cores are used. If $\leqslant$ 10 points are requested in all, only master samples.
- Typically no hardware limit for p_{cores} but useful for load-levelling.
- Moderate value for p_{cores} (e.g. 10 000) may improve performance unless integrand is known to evaluate equally fast everywhere.

Controlling Parallelization

- Accelerators are set via environment

  ```
  CUBAACCEL=n_accel           (default: 0)
  CUBAACCELMAX=p_accel        (default: 1000)
  ```

  CUBAACCEL=n_{accel} (default: 0)
  CUBAACCELMAX=p_{accel} (default: 1000)

  or API call

  call cubaaccel(n_{accel}, p_{accel})

- Cores are set via environment

  CUBACORES=n_{cores} (default: no. of idle cores)
  CUBACORESMAX=p_{cores} (default: 10 000)

  or API call

  call cubacores(n_{cores}, p_{cores})

Spinning Cores

- Workers usually **started and stopped automatically.** User can start them manually or keep them running.
- Start workers with `cubafork`, shut down with `cubawait`.
- Running workers **will not 'see' subsequent changes** in master's data (`common`) or code (`dlsym`).
  (Can of course arrange with shared memory etc.)
- Keep cores running:

```
void *spin = NULL;

Vegas(..., &spin, ...);
...
cubawait(&spin);
```

Manually start cores:

```
void *spin;
cubafork(&spin);
Vegas(..., &spin, ...);
...
cubawait(&spin);
```

- Controlled through **'Spinning Cores' pointer.**

(De)Initialization of Workers

- Register init/exit functions with

```
cubainit(initfun, initarg);
cubaexit(exitfun, exitarg);
```

- Will be called as

```
initfun(initarg, &core);
exitfun(exitarg, &core);
```

  where `core` has same meaning as in integrand:
  `core` $<$ 0: Accelerator, $\geqslant$ 0: Core, = 32768: Master.

- Executed on worker after `fork`/before `wait` (always),
  on master only when sampling is done.

- For Accelerators typically used to set up the GPU for the integrand evaluations.

Vectorization

- Vectorization = evaluate integrand for several points at once (SIMD).
- Vector instructions commonly available (SSE, AVX).
- Cuba does not automatically vectorize integrand.
- Cuba can pass more than one point (`nvec`) per integrand invocation.
- `nvec` need not correspond to hardware vector length - can make sense e.g. if computations have significant intermediate results in common.

Concurrency Issues

- `fork` creates independent process image.
- Cannot easily communicate back results other than the intended output to the master.
- Cannot easily communicate between workers.
- `fork` does not guard other common resources, e.g. files.
- If integrand writes to file, output may be 'chaotic'.
  No buffered output.
  Better: each worker writes to own file.

Speed-ups

Assess parallelization efficiency through

$$\text{speed-up} = \frac{t_{\text{serial}}}{t_{N\text{-cores}}} \quad \text{ideally} = N.$$

- Parallelization overhead = Extra time for communication, scheduling efficiency etc.
  Overhead can be estimated through $t_{\text{serial}}/t_{\text{1-core}} < 1$.
- Load-levelling = Keeping cores busy. If only $N - n$ busy, absolute timing may be ok but N-core speed-up lousy.
- Caveat: Hyperthreading, e.g. i7 has 8 virtual, 4 real cores.

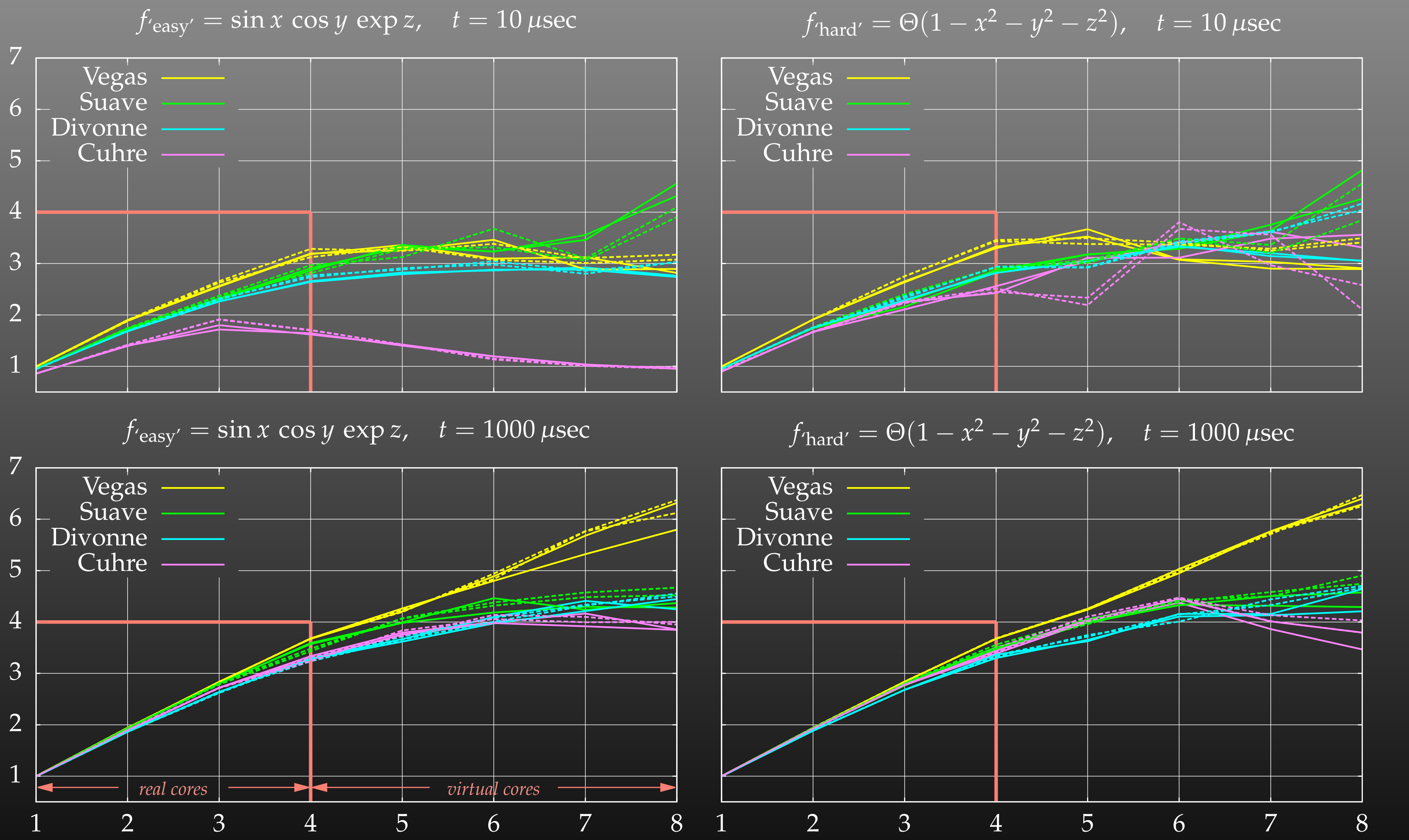
Speed-ups will obviously depend on the 'cost' of the integrand: The more time a single integrand evaluation takes, the better speed-ups can be expected to achieve.

Timing Measurements

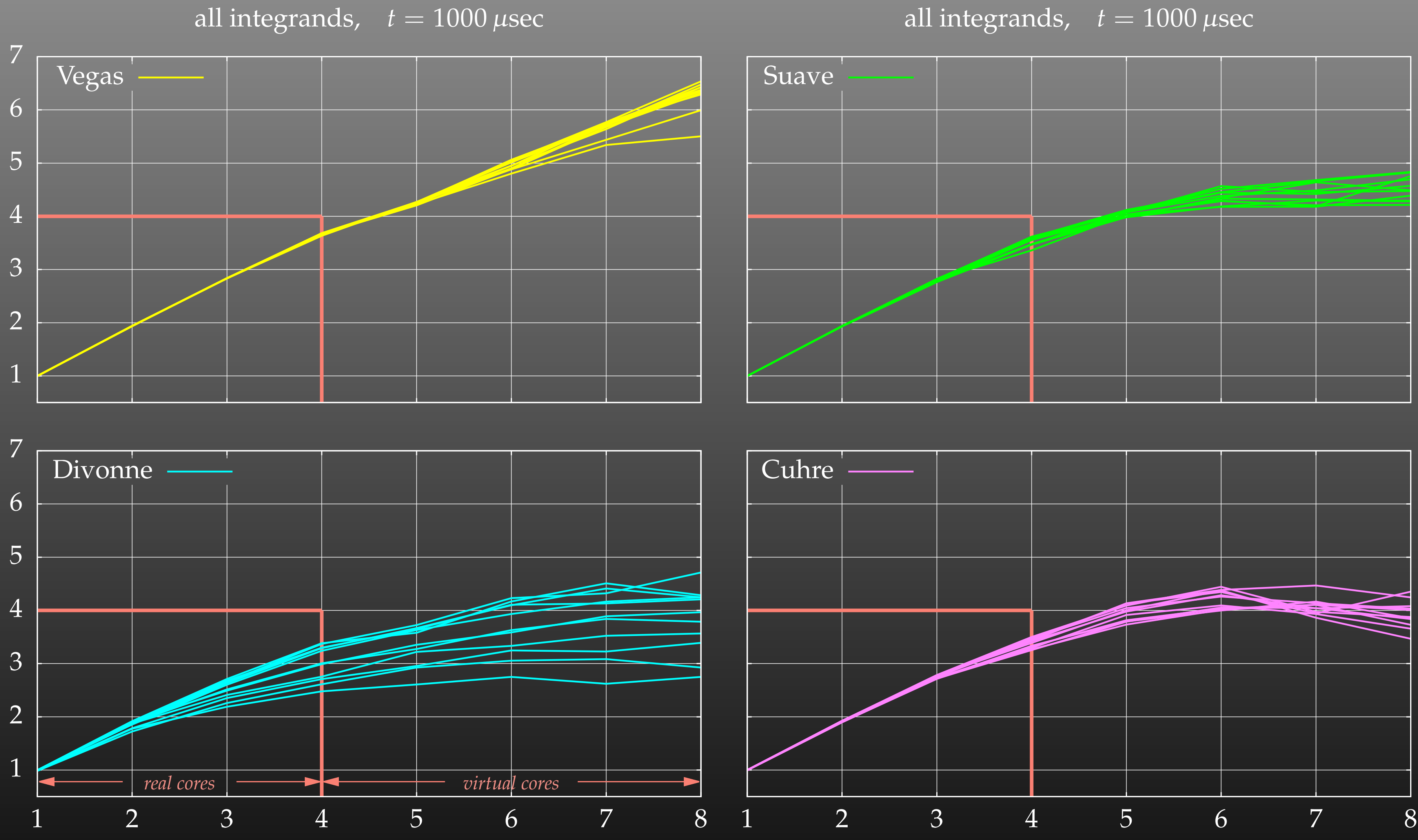
Timing measurements delicate on multicore systems:

- System timer (even `ualarm`) has granularity.
- Cannot use timer interrupt directly in integrand delay, accumulates too large errors.
- First calibrate delay loop over sufficiently long time interval.
- Use same calibrated value per machine for all runs.
- Repeat integrations such that each measurement takes a reasonable minimum amount of time (to minimize measurement errors).
- Disable processes like `condor_start`, `autonice`, etc.

Timings: ‘easy’ vs ‘hard’



Timings: all integrands



Résumé

- Cuba now features concurrent sampling.
- Achieves significant speed-ups.
- No extra software needs to be installed.
- No reentrant integrand required.
- Parallelization is switched on automatically, can be controlled through environment, API calls.
- More details in arXiv:1408.0663.

BACKUP SLIDES

Integrand Functions in the Result Plots

$$f_1 = \sin x \, \cos y \, \exp z, \qquad f_2 = \frac{\cos y \, \exp z}{(x+y)^2 + .003},$$

$$f_3 = \frac{1}{3.75 - \cos(\pi x) - \cos(\pi y) - \cos(\pi z)},$$

$$f_4 = |x^2 + y^2 + z^2 - .125|, \qquad f_5 = \exp(-x^2 - y^2 - z^2),$$

$$f_6 = \frac{1}{1 - xyz + 10^{-10}}, \qquad f_7 = \sqrt{|x - y - z|},$$

$$f_8 = \exp(-xyz), \qquad f_9 = \frac{x^2}{\cos(x + y + z + 1) + 5},$$

$$f_{10} = \begin{cases} x > \frac{1}{2} & \dfrac{1}{\sqrt{xyz + 10^{-5}}}, \\ \text{else} & \sqrt{xyz} \end{cases} \qquad f_{11} = \Theta(1 - x^2 - y^2 - z^2).$$

Deterministic vs. Monte Carlo Methods

Deterministic

Use a **Quadrature Formula**

$$\mathrm{I}f \approx \mathrm{Q}_n f := \sum_{i=1}^{n} w_i f(\vec{x}_i)$$

with specially chosen **Nodes** $\vec{x}_i$ and **Weights** w_i.

Error estimation e.g. by **Null Rules** N_m which give zero for functions Q_n integrates exactly and thus measure errors due to “higher terms.”

Monte Carlo

Take the **Statistical Average** over random samples $\vec{x}_i$

$$\mathrm{I}f \approx \mathrm{M}_n f := \frac{1}{n} \sum_{i=1}^{n} f(\vec{x}_i) \,.$$

The **Standard Deviation** is a probabilistic estimate of the integration error:

$$\sigma(\mathrm{M}_n f) = \sqrt{\mathrm{M}_n f^2 - \mathrm{M}_n^2 f} \,.$$

Globally Adaptive Subdivision

If an error estimate is available, global adaptiveness is easy to implement:

- Integrate the entire region: $I_{\text{tot}} \pm E_{\text{tot}}$.
- while $E_{\text{tot}} > \max(\varepsilon_{\text{rel}} I_{\text{tot}}, \varepsilon_{\text{abs}})$
-     Find the region r with the largest error.
-     Bisect (or otherwise cut up) r.
-     Integrate each subregion of r separately.
-     $I_{\text{tot}} = \sum I_i,\ E_{\text{tot}} = \sqrt{\sum E_i^2}$.
- end while

Importance Sampling

In **Importance Sampling** one introduces a weight function:

$$\mathbf{I}f = \int_0^1 \mathrm{d}^d x \, w(\vec{x}\,) \, \frac{f(\vec{x}\,)}{w(\vec{x}\,)} \,, \qquad w(\vec{x}\,) > 0 \,, \quad \mathbf{I}\, w = 1 \,.$$

- One must be able to sample from the distribution $w(\vec{x}\,)$,
- f/w should be "smooth," such that $\sigma_w(f/w) < \sigma(f)$, e.g. w and f should have the same peak structure.

The ideal choice is known to be $w(\vec{x}\,) = |f(\vec{x}\,)|/\mathbf{I}f$ which has $\sigma_w(f/w) = 0$.

Example: Vegas uses piecewise constant weight funct (grid).

Stratified Sampling

Stratified Sampling works by sampling subregions. Consider:

	n samples in total region $r_a + r_b$	$n_a = n/2$ samples in r_a, $n_b = n/2$ samples in r_b
Integral	$\mathbf{I}f \approx \mathbf{M}_n f$	$\mathbf{I}f \approx \frac{1}{2}(\mathbf{M}^a_{n/2} f + \mathbf{M}^b_{n/2} f)$
Variance	$\frac{\sigma^2 f}{n}$ $= \frac{1}{2n}(\sigma_a^2 f + \sigma_b^2 f) + \frac{1}{4n}(\mathbf{I}_a f - \mathbf{I}_b f)^2$	$\frac{1}{4}\left(\frac{\sigma_a^2 f}{n/2} + \frac{\sigma_b^2 f}{n/2}\right)$ $= \frac{1}{2n}(\sigma_a^2 f + \sigma_b^2 f)$

The optimal reduction of variance is for $n_a/n_b = \sigma_a f/\sigma_b f$.
Thus: Split up the integration region into parts with equal variance, then sample all parts with same number of points.
But: naive splitting causes a 2^d increase in regions!

Number-Theoretic Methods

The basis for the number-theoretical formulas is the **Koksma-Hlawka Inequality:**

The error of every $\mathrm{Q}_n f = \frac{1}{n}\sum_{i=1}^n f(\vec{x}_i)$ is bounded by

$$|\mathbf{Q}_n f - \mathbf{I} f| \leqslant V(f)\, D^*(\vec{x}_1, \dots, \vec{x}_n)\,.$$

where V is the **"Variation in the sense of Hardy and Krause"** and D^* is the **Discrepancy** of the sequence $\vec{x}_1, \dots, \vec{x}_n$,

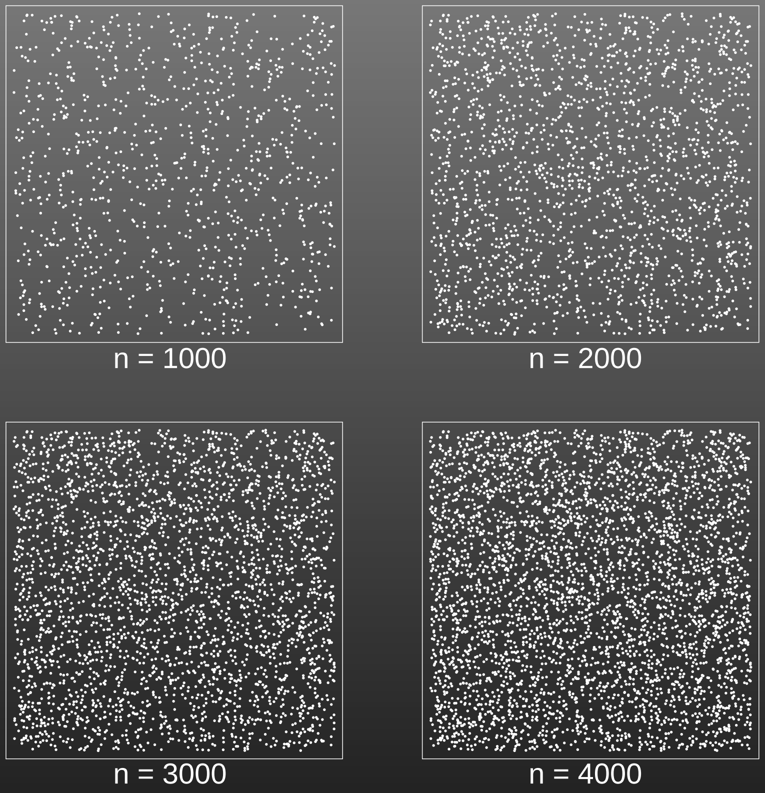
$$D^*(\vec{x}_1, \dots, \vec{x}_n) = \sup_{r \in [0,1]^d} \left| \frac{\nu(r)}{n} - \mathrm{Vol}\, r \right|,$$

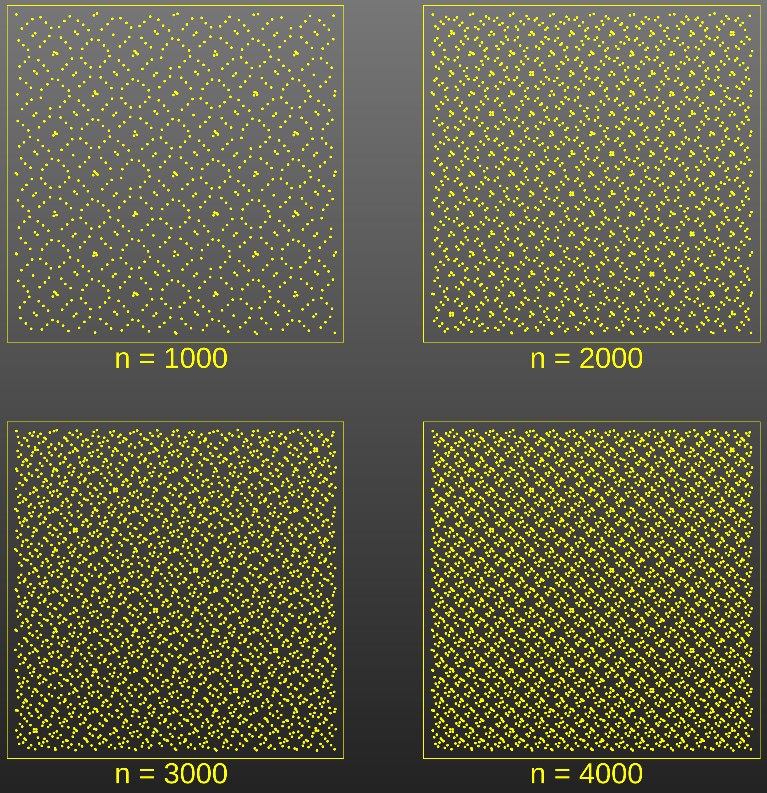
where $\nu(r)$ counts the $\vec{x}_i$ that fall into r.
For an **Equidistributed Sequence,** $\nu(r) \propto \mathrm{Vol}\, r$.

Comparison of Sequences

Mersenne Twister
Pseudo-Random Numbers

Sobol
Quasi-Random Numbers





$$\mathcal{O}(1/\sqrt{n})$$

$$\mathcal{O}(\log^{d-1} n/n)$$