An experience report on (auto-)tuning of mesh-based PDE solvers on shared memory systems

Dominic E. Charrier, Tobias Weinzierl

With the advent of manycore systems, shared memory parallelisation has gained importance in high performance computing. Once a code is decomposed into tasks or parallel regions, it becomes crucial to identify reasonable grain sizes, i.e. minimum problem sizes per task that make the algorithm expose a high concurrency at low overhead. Many papers do not detail what reasonable task sizes are, and consider their findings craftsmanship not worth discussion. We have implemented an autotuning algorithm, a machine learning approach, for a project developing a hyperbolic equation system solver. Autotuning here is important as the grid and task workload are multifaceted and change frequently during runtime. In this paper, we summarise our lessons learned. We infer tweaks and idioms for general autotuning algorithms and we clarify that such a approach does not free users completely from grain size awareness.

Durham University
School of Engineering and Computing Sciences

An experience report on (auto-)tuning of mesh-based PDE solvers on shared memory systems

PPAM17

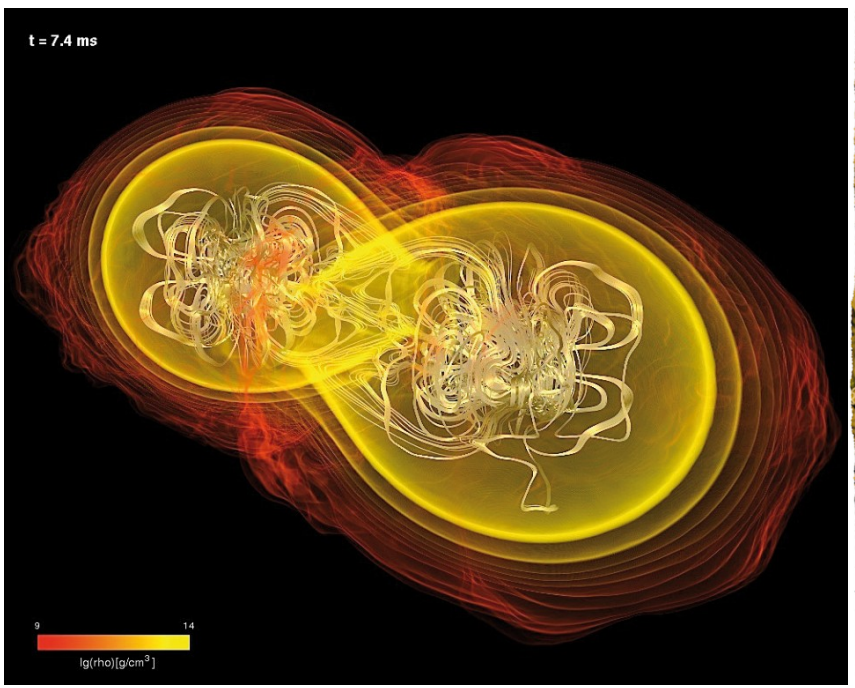
September 2017

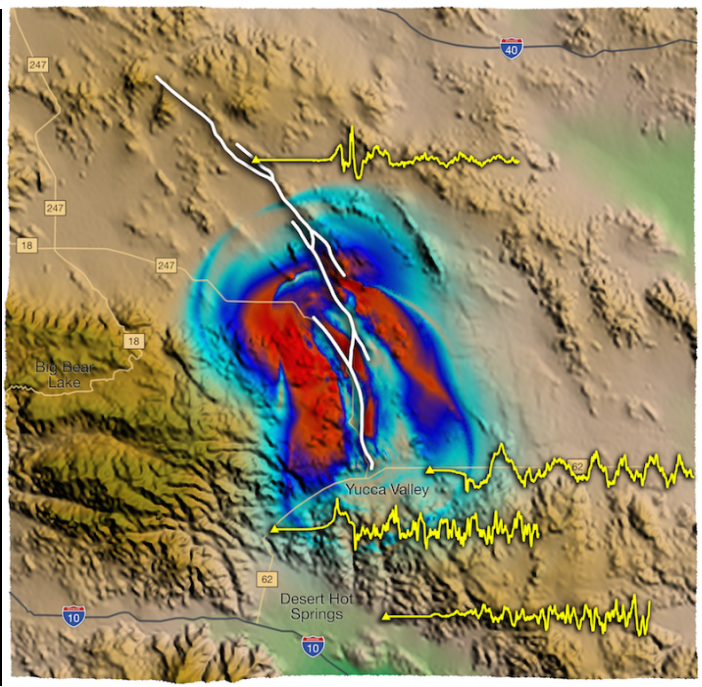
The project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671698 (ExaHyPE).

An Exascale Hyperbolic PDE Engine

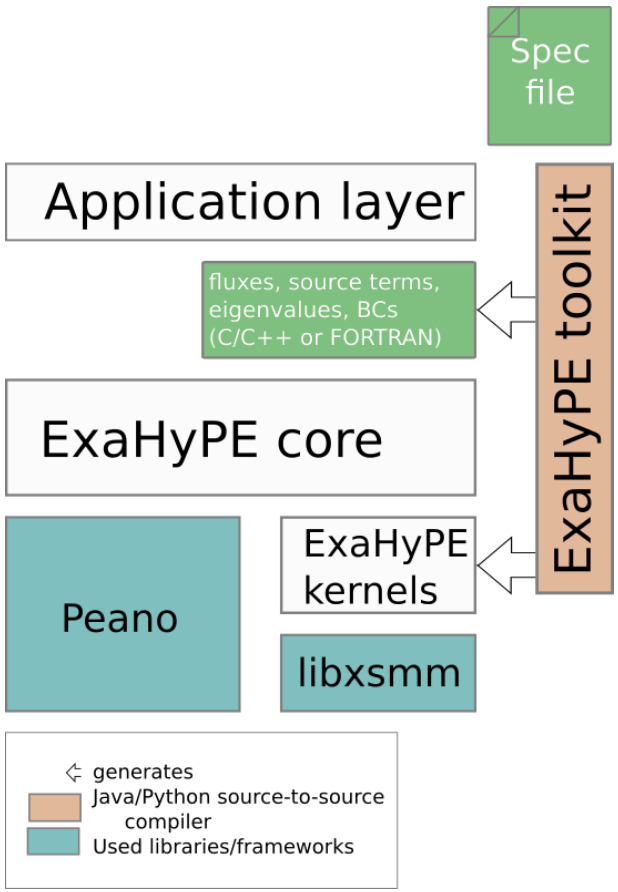
Durham University
School of Engineering and Computing Sciences

- One simulation engine
  Similar to a 3D game engine
- Enable groups to write an exascale code within a year
  No extreme scale expertise required but some HPC affinity
- Two grand challenges
  Seismic risk assessment and gravitational waves





Software architecture and usage



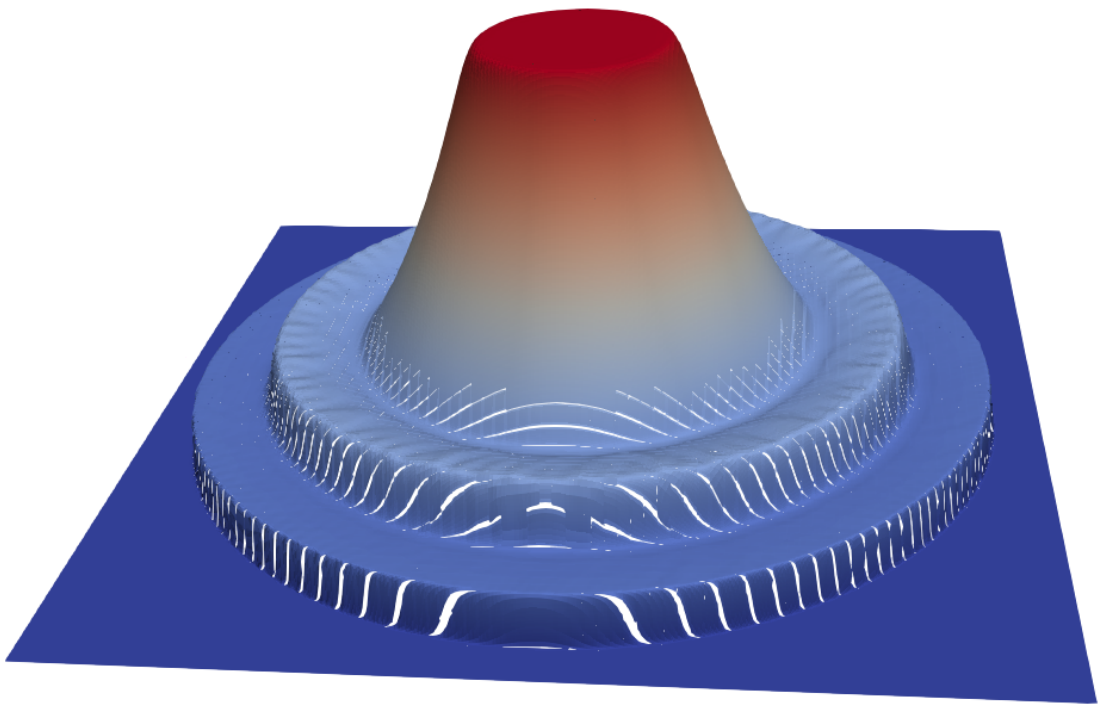
- First-order hyperbolic PDEs
  $\partial_t u + \nabla \cdot F + \sum_i B_i \frac{\partial u}{\partial x_i} = S$
- ADER-DG with Finite Volumes limiter on adaptive Cartesian meshes
- User code focuses on model/math
  "What" is done not "how"
- User code integration and generation of tailored kernels via precompiler
  Vectorisation and shared memory efficiency
- Architecture provides efficiency and parallelism
  MPI+TBB / MPI+OpenMP / MPI+TBB+CUDA

Durham University
School of Engineering and Computing Sciences

Example: Solving the Euler equations

- Compressible Euler equations in conservation form:

$$\partial_t \begin{pmatrix} \rho \\ \mathbf{j} \\ E \end{pmatrix} + \nabla \cdot \begin{pmatrix} \mathbf{j} \\ \frac{1}{\rho}\mathbf{j} \otimes \mathbf{j} + p \cdot \mathbf{I} \\ \frac{1}{\rho}(E + p) \cdot \mathbf{j} \end{pmatrix} = 0, \qquad p = (\gamma - 1) \cdot \left(E - 0.5 \frac{1}{\rho} \mathbf{j} \cdot \mathbf{j} \right)$$



ADER-DG with a-posteriori limiting

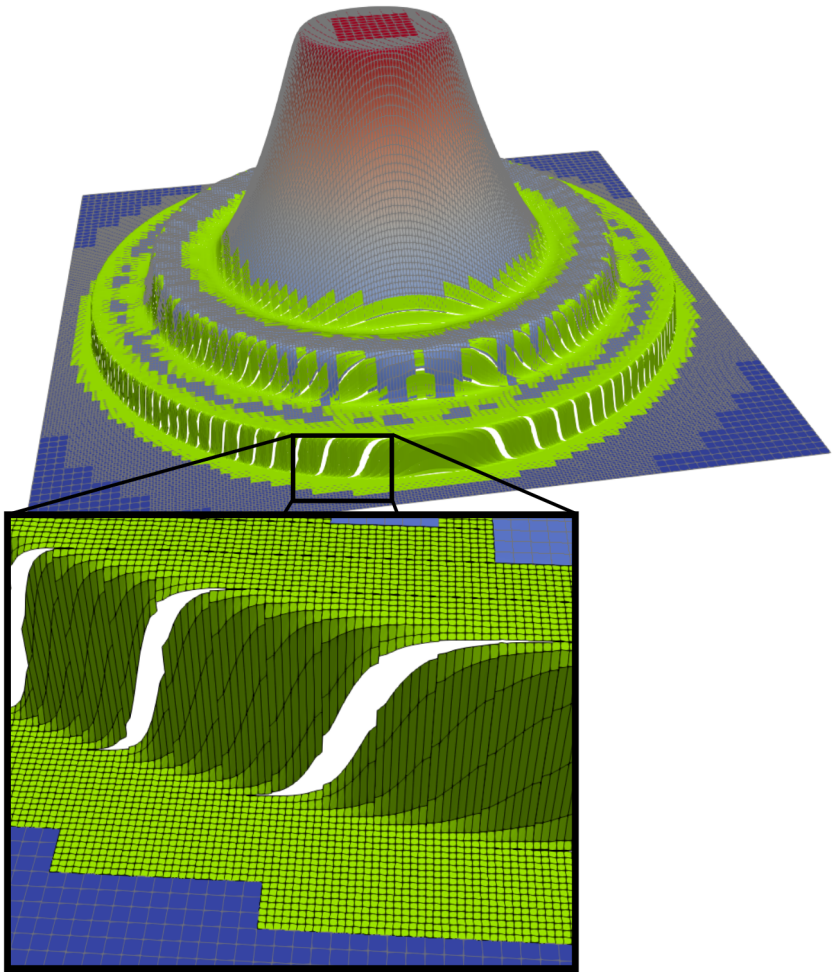
Durham University
School of Engineering and Computing Sciences

- Algorithmic steps (tasks):

`STP`	Cellwisely solve implicit problem via Picard iterations: $\int_I \int_K (\partial_t q_h + \nabla \cdot F)\ \varphi_h \, \mathrm{d}x\mathrm{d}t = 0$
`Riemann`	Facewisely determine numerical normal flux $\mathbf{G}(q_h{}^+, q_h{}^-)\mathbf{n}$
`Update`	Cellwisely evolve using volume and face integral contributions: $(D_h, v_h)_K = -(\mathbf{F}(q_h), \nabla v_h)_{(K\times I)} + (\mathbf{G}(q_h{}^+, q_h{}^-)\mathbf{n}, v_h)_{(\partial K\times I)}$

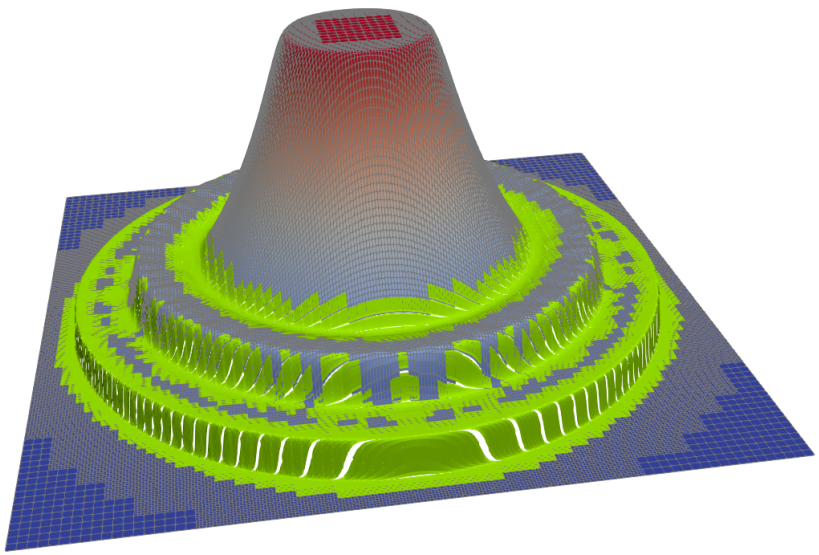
- Further tasks may be introduced
  Non-physical oscillations are cured a-posteriori with robust FV
  Calculation of time step size
- Facewise `Riemann` solves synchronise neighbouring cells
- `Update` plus `STP` are embarrassingly concurrent

Challenges



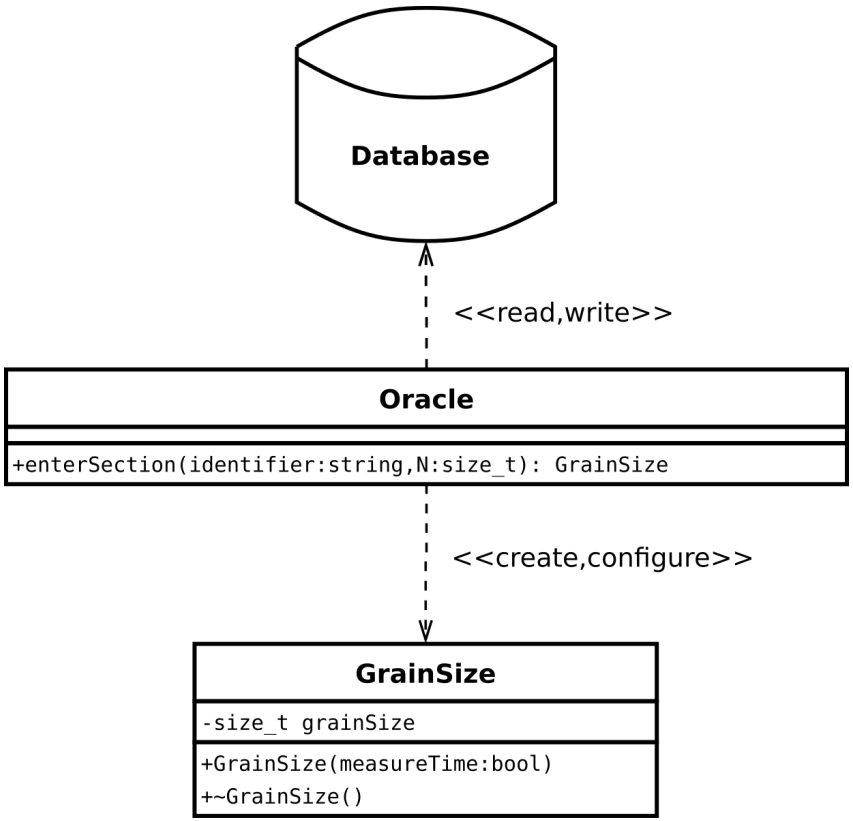
- Runtime of some tasks varies
  - Cell solution is evolved using ADER-DG or FV
  - STP Picard iterations differ from cell to cell
- Changing task dependency patterns
  - Dynamic adaptive mesh refinement
  - Solution recomputation with FV
- Tasks have different characteristics
  - Bandwidth-bound vs. compute-bound
- Machine, PDE, and approximation quality change task characteristics

Outline



An autotuning algorithm

Durham University
School of Engineering and Computing Sciences



- Central instance (singleton) `Oracle` manages database
- Code runs through grid notifies `Oracle` about code section it is about to enter plus problem size *N*
- `Oracle` returns `GrainSize` instance. `GrainSize` can be configured to return measured lifetime upon destruction
- Using GrainSize objects enables to work with nested parallel sections

Recorded data

Durham University
School of Engineering and Computing Sciences

- Database record:

`codeSection`	identifier for the code section (key)
$N_{\max}$	maximum problem size associated with `codeSection`
g	grain size used for this problem ($g = N_{\max}$ means that parallelisation does not pay off)
Δg	delta w.r.t the previous g ($g + \Delta g < N_{\max}$)
t_s	serial runtime (runtime without any parallelisation)
t_g	runtime using (current) g

- `Oracle` adds new entry every time no record is found for code section or $N > N_{\max}$
- Grain size g is initialised as either $g = N/2$ or N/p for p threads
  Initial guess depends on N itself (see next slide)

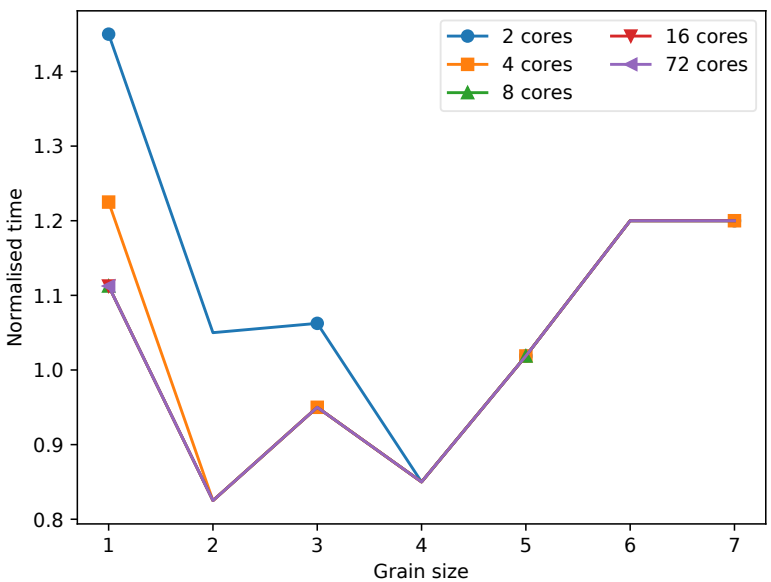
Performance model

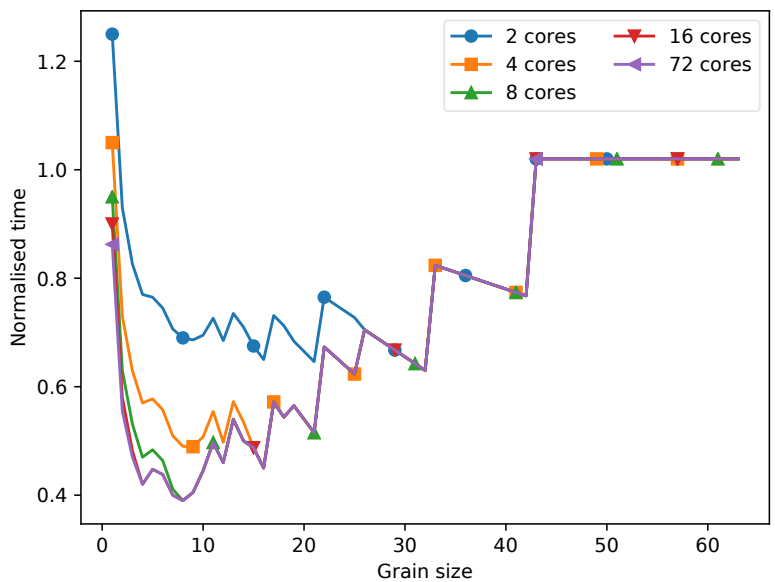
- We extend Amdahl's law by task administration overhead $h \propto p$:

$$t_g = (1 - \hat{f}) \cdot \frac{t_s}{min\left(\left\lfloor \frac{N}{g} \right\rfloor, p\right)} + \hat{f} \cdot t_s + h \cdot \left\lceil \frac{N}{g} \right\rceil \text{ with } \hat{f} = f + \frac{N \bmod g}{N}(1 - f)$$

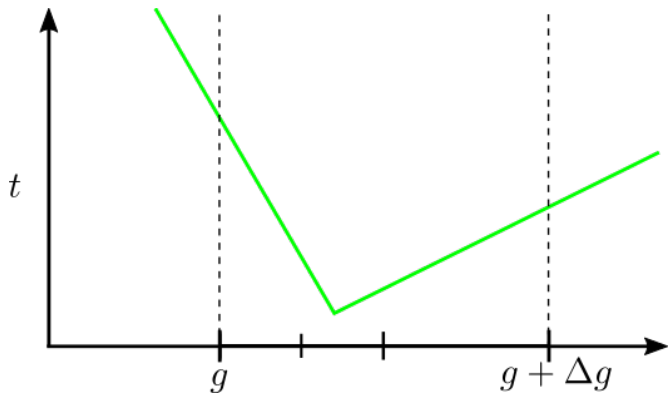
$f \in [0, 1]$: genuinely serial code sections

$\Rightarrow$ Model motivates initial choice of $g = \frac{N}{2}$ for small N (left: $N = 8$) and $g = \frac{N}{p}$ for large N (right: $N = 64$)

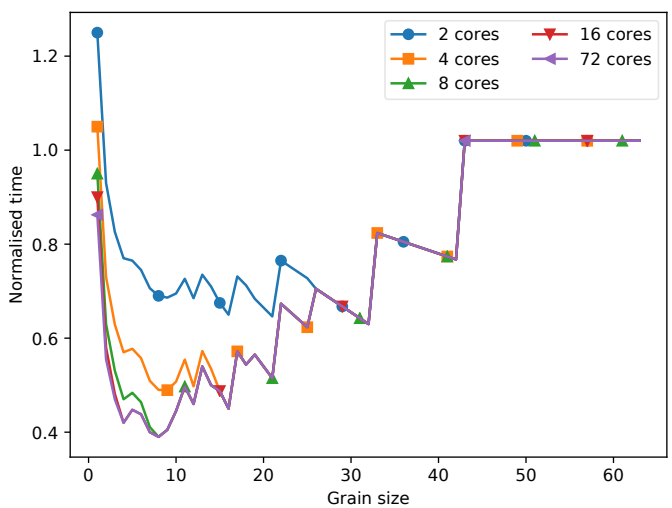




Algorithmic idea



Finding minima by interval halving



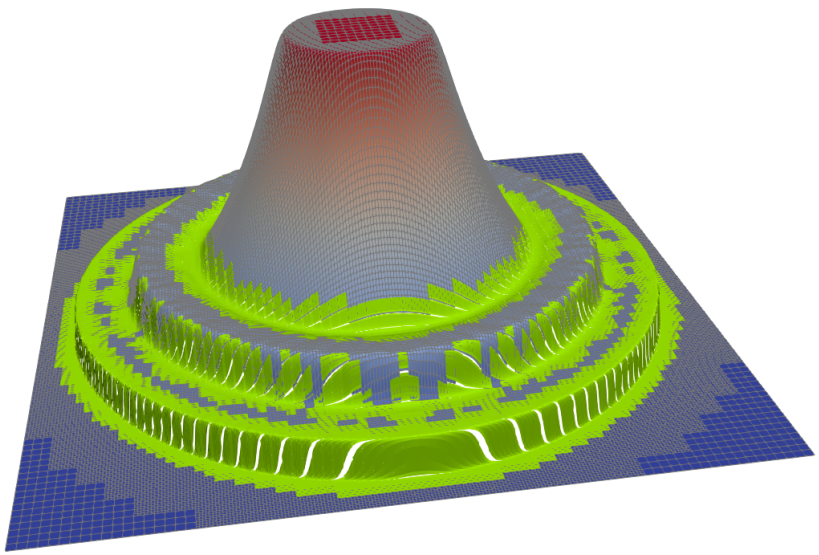
Performance model for $N = 64$

- Autotuning algorithm works with "omni-present" parallelisation
- Parallelisation is turned off where it does not pay off
- Search good grain sizes g only for remaining code sections, e.g. by interval halving

  Shrink g with steps Δg until runtime rises again

  Then, fall back to previous g and use $\Delta g/2$
- Frequent restarts for avoiding local minima

Outline



Taking timings

Durham University
School of Engineering and Computing Sciences

- Timings are subject to noise. `Oracle` thus tracks averaged times $\langle t_g \rangle$
- New timing t_g for code section is valid if $|\langle t_g \rangle^{new} - \langle t_g \rangle^{old}| < \varepsilon$

Implementation pitfall (Linux timer invocation overhead)

Linux timer invocations come with overhead which quickly pollutes timings

$\Rightarrow$ Perform measurements only in one code section per grid sweep

Implementation pitfall (Measuring the serial runtime first)

- *All timings have to converge subject to ε.*
- *If we determine t_s first, it takes a long time until any parallelisation is enabled at all. This is not acceptable in HPC*

$\Rightarrow$ Randomise grain size choice whenever measurements for section are taken

One out of $\frac{N_{\max}}{g}$ samples measure serial runtime t_s

Otherwise, t_g is measured

Binning

Durham University
School of Engineering and Computing Sciences

Usage pitfall (Directly extrapolate grain sizes to larger problems)

For our use-case, we cannot assume a linear relationship between g and N.

$\Rightarrow$ Track good grain sizes per problem size. We use a binning approach

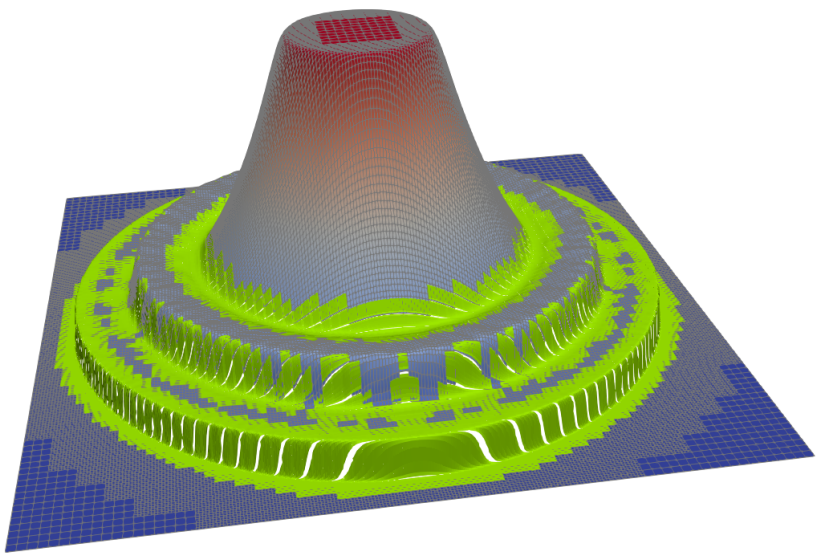
- If we encounter new code section (with size N), we initialise bins

$$
\begin{aligned}
2^1 &< N_{\max} \leq 2^2 \\
2^2 &< N_{\max} \leq 2^4 \\
&\vdots \\
2^{k-1} &< N_{\max}, N \leq 2^k
\end{aligned}
$$

- If next smaller bin exists, we initialise newly added "larger" bin with extrapolated g
- If "small" bin converged, we extrapolate its g to all not yet converged "larger" ones

We list five more pitfalls in our paper...

Outline



History vs. context

Durham University
School of Engineering and Computing Sciences

Observation (Context-aware autotuning is mandatory)

- *Our code reacts sensitively to machine type, core count, input data sets*
- *Some problem setups perform poorly with autotuning settings derived for others*

$\Rightarrow$ Perform autotuning searches per problem setup

$\Rightarrow$ Don't use central database for all setups

Observation (Accuracy improves over time)

The more samples, the more reliable the measurement data

$\Rightarrow$ Per problem, we store/load autotuning parameters after/before each simulation

We persist the database

- Simulations can continue learning or apply loaded parameters

Autotuning for large HPC runs

Durham University
School of Engineering and Computing Sciences

Observation (Autotuning is problematic for large HPC runs)

It is important to search autotuning parameters on target machine.
However, it is problematic to obtain such parameters for large HPC runs:

- *Autotuning runs temporarily into inefficient parameter choices*
- *Autotuning overhead must be multiplied by number of nodes*
- *Single-node parameter studies might be deemed unsuitable*

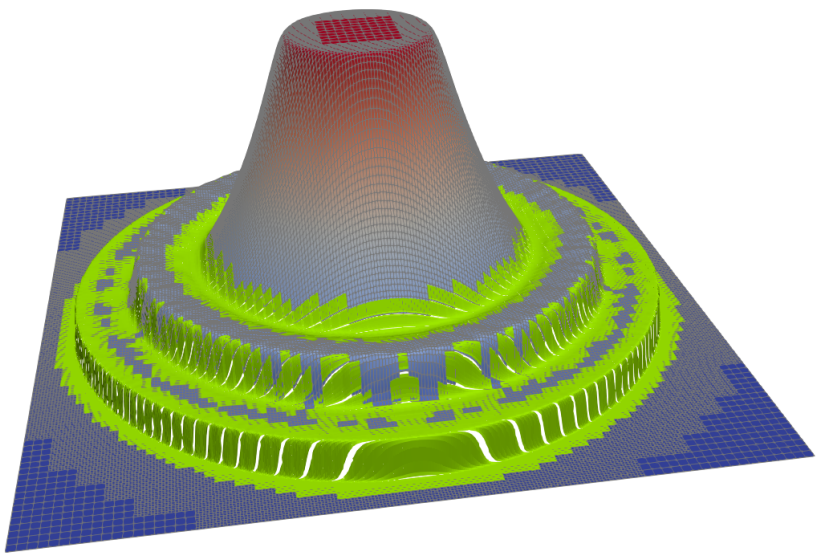
$\Rightarrow$ We thus augment our binning. We run small-scale, yet characteristic runs briefly, and extrapolate reasonable grain sizes to large production runs

$\Rightarrow$ We sacrifice only a single node per experiment to perform the parameter search

The node dumps its new knowledge into the parameter file

Other nodes read from the file at startup

Outline



Computational evidence

Durham University
School of Engineering and Computing Sciences

- Compared autotuning strategies:

`autotuning-with-finest`	runs the autotuning strategy without taking existing records into account
`autotuning-from-coarse-grid`	runs cascade of autotuning experiments. Learns on more and more finer meshes
`autotuning-from-coarse-grid-without-learning`	Takes the final dump of `autotuning-from-coarse-grid` and runs with learning switched off

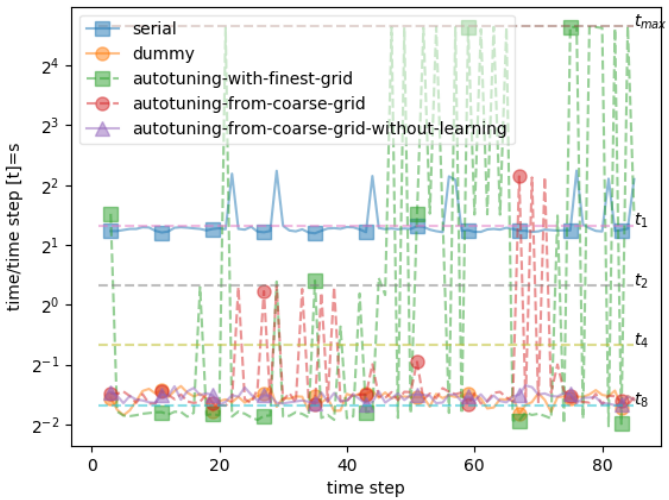
- We further compare against:

`serial`	provides the baseline and normalises runtimes
`dummy`	manually tuned for good results

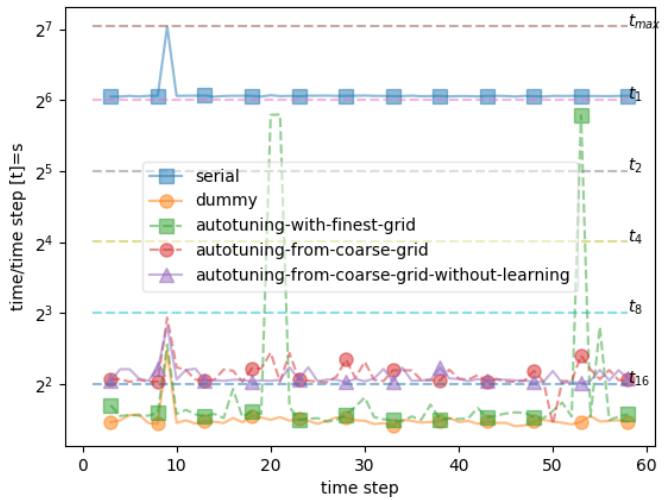
- Haswell Xeon E5-2697 with 28 cores and 2.6 GHZ base clock
- Implementation relies on Intel's TBB

A smooth solution of the Euler equations

Durham University
School of Engineering and Computing Sciences



Low approximation order & arithmetic intensity

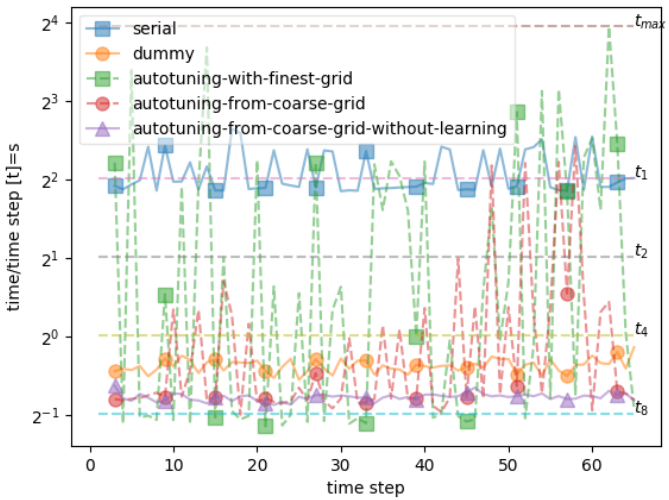


High approximation order & arithmetic intensity

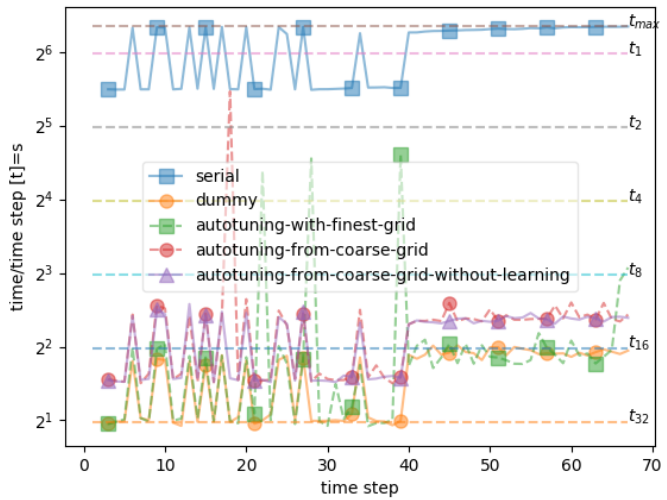
- We employ pure ADER-DG (grid is uniform)
- Autotuning works from first iteration on
- `with-finest-grid` suffers from runtime spikes
- `from-coarse-grid` vs `without-learning` shows price for sliding updates of t_s
- Cascading removes spikes however might yield suboptimal results (right plot)

A discontinuous solution of the Euler equations

Durham University
School of Engineering and Computing Sciences



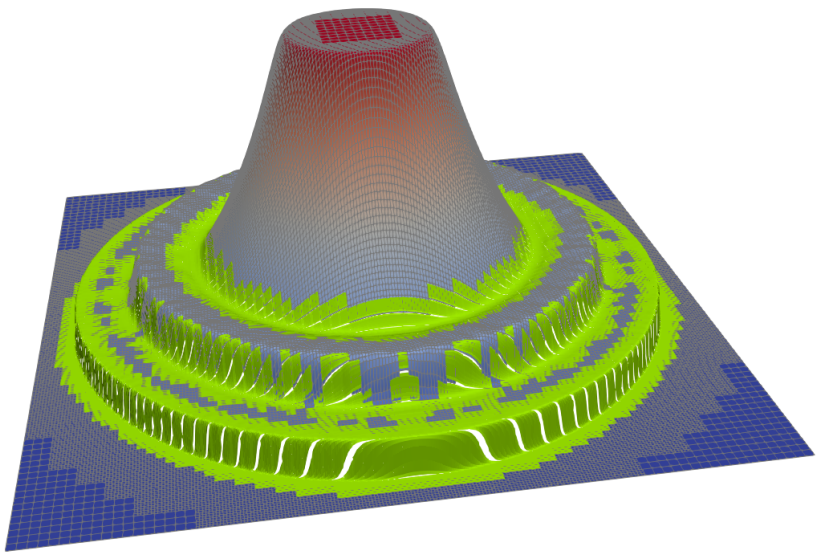
Low approximation order & arithmetic intensity



High approximation order & arithmetic intensity

- ADER-DG is now coupled to FV (grid is uniform)
  Serial runtime is now very dynamic, too
- Mostly similar individual behaviour of strategies
- `from-finest-grid` struggles for low order solve
- Other strategies seem more robust

Outline



Summary

Durham University
School of Engineering and Computing Sciences

- We proposed a blackbox autotuning strategy for codes with omni-present parallelisation
- Our algorithm turns off parallelisation first where it does not pay off.
  It tries to find good grain sizes for the remaining code sections
- Considered autotuning strategies could compete with laborious, manual grain size choice for well-behaved problems
- Our use-case, an ExaHyPE application, comes with challenges which require awareness of the user despite the initial blackbox idea

  Hard to predict task runtime and dependencies

  Binning and extrapolating grain sizes yield a more robust overall strategy

Next steps

- For many setups, our autotuning reduces the number of employed cores. Other MPI ranks (on same node) could grab these freed cores

  Invasive computing

Links

http://exahype.eu/exahype-engine

http://www.peano-framework.org

Support & acknowledgements

- This talk picks up challenges tackled by the project ExaHyPE (www.exahype.eu). The project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671698 (ExaHyPE).

- The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC at Leibniz Supercomputing Centre (LRZ, www.lrz.de).

- Experiments were made possible through Durham's supercomputer Hamilton.