The ExaStencils DSL ExaSlang

Christian Schmitt[1], Stefan Kronawitter[2], Sebastian Kuckuk[3], Frank Hannig[1], Jürgen Teich[1], Harald Köstler[3], Ulrich Rüde[3], Christian Lengauer[2]
[1] Hardware/Software Co-Design, Friedrich-Alexander Universität Erlangen-Nürnberg (FAU)
[2] Chair of Programming, University of Passau
[3] System Simulation, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

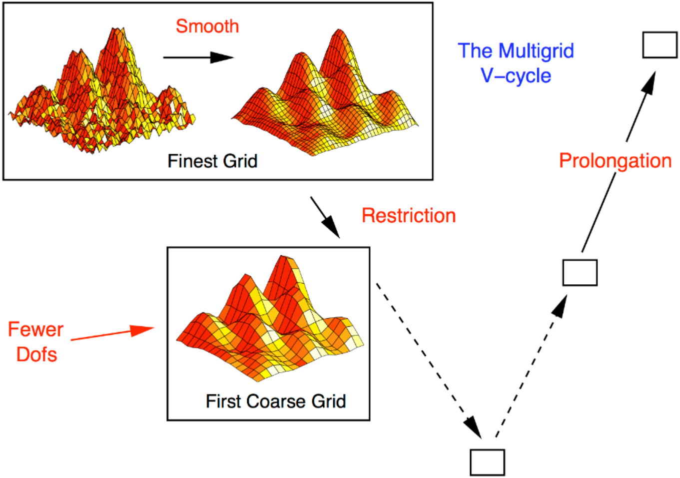
IFIP WG 2.11 Meeting, London, 9-12 November 2015

Christian Schmitt et al. "ExaSlang: A Domain-Specific Language for Highly Scalable Multigrid Solvers". In: *Proceedings of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. (New Orleans, LA, USA). IEEE Computer Society, Nov. 17, 2014, pp. 42–51

A Multigrid Language

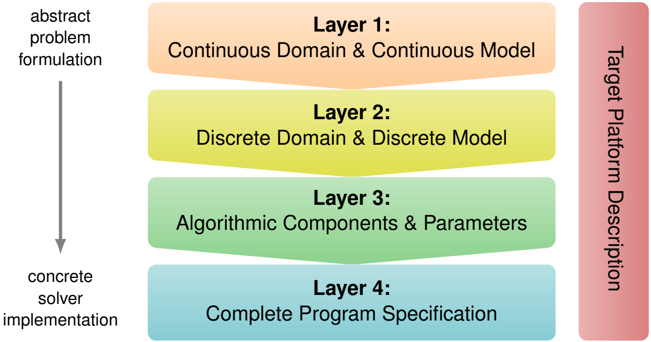
ExaSlang: ExaStencils language

Domain: massively parallel geometric Multigrid solvers



ExaSlang: a multi-layered, external DSL

Different layers of ExaSlang address different users and knowledge.



- Syntax borrowed from Scala
- Parsing and code transformation framework implemented in Scala[1]

[1] Christian Schmitt et al. "An Evaluation of Domain-Specific Language Technologies for Code Generation". In: *Proceedings of the 14th International Conference on Computational Science and its Applications (ICCSA)*. (Guimaraes, Portugal). IEEE Computer Society, June 30–July 3, 2014, pp. 18–26

ExaSlang: Layers

Continuous Domain & Continuous Model (Layer 1)

Specification of

- size and structure of computational domain
- variables
- functions and operators (pre-defined functions and operators also available)
- partial differential equation
- mathematics, typically on paper or in $\LaTeX$

ExaSlang: Layers

Continuous Domain & Continuous Model (Layer 1)

Specification of

- size and structure of computational domain
- variables
- functions and operators (pre-defined functions and operators also available)
- partial differential equation
- mathematics, typically on paper or in $\LaTeX$

Discrete Domain & Discrete Model (Layer 2)

Discretization of

- computational domain into fragments (e.g., triangles)
- variables to fields (multi-dimensional arrays)
  - specification of data types
  - choice of discretization strategy

ExaSlang: Layers

Algorithmic Components & Parameters (Layer 3)

Specification of

- discretized mathematical operators
- multigrid components (e.g., choice of smoother)
- operations in matrix notation

ExaSlang: Layers

Algorithmic Components & Parameters (Layer 3)

Specification of

- discretized mathematical operators
- multigrid components (e.g., choice of smoother)
- operations in matrix notation

Complete Program Specification (Layer 4)

Specification of

- complete multigrid V-cycle, or custom cycle types
- operations depending on the multigrid level
- loops across computational domain
- communication and data exchange
- interface to third-party code

ExaSlang 4: Complete Program Specification

Properties

- Procedural
- Statically typed
- External domain-specific language
- Syntax largely inspired by Scala

```
Function JacobiSmoother@((coarsest + 1) to finest)() : Unit {
  communicate ghost of Solution[active]@current
  loop over fragments {
    loop over Solution@current {
      Solution[next]@current = Solution[active]@current
        + (omega * inverse(diag(Laplace@current))
          * (RHS@current - Laplace@current
            * Solution[active]@current))
    }
    advance Solution@current
  }
}
```

ExaSlang 4: Complete Program Specification

Properties

- Procedural
- Statically typed
- External domain-specific language
- Syntax largely inspired by Scala

```
Function JacobiSmoother@((coarsest + 1) to finest)() : Unit {
  communicate ghost of Solution[active]@current
  loop over fragments {
    loop over Solution@current {
      Solution[next]@current = Solution[active]@current
        + (omega * inverse(diag(Laplace@current))
          * (RHS@current - Laplace@current
            * Solution[active]@current))
    }
    advance Solution@current
  }
}
```

ExaSlang 4: Level Specifications

Multigrid is inherently hierarchical and recursive

➔ We need

- an exit condition for the multigrid recursion
- at any one level, access to the data and functions at other levels

➔ Additionally, we would like

- relative addressing
- level-specific aliases
- level-specific variable definitions

Implementation

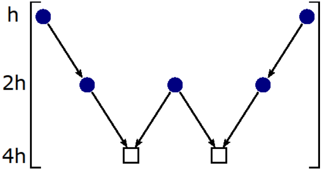
- Level numbers, e.g., `@0` for bottom level
- Aliases, e.g., `@all`, `@current`, `@coarser`, `@coarsest`
- Simple expressions, e.g., `@(coarsest + 1)`
- Ranges, e.g., `@(1, 3, 5)`, `@(1 to 5)`, `@(1 to 5, not(3))`
- Variables, e.g., `i@all`, `x@0`

ExaSlang 4: Example

Example: exit multigrid recursion

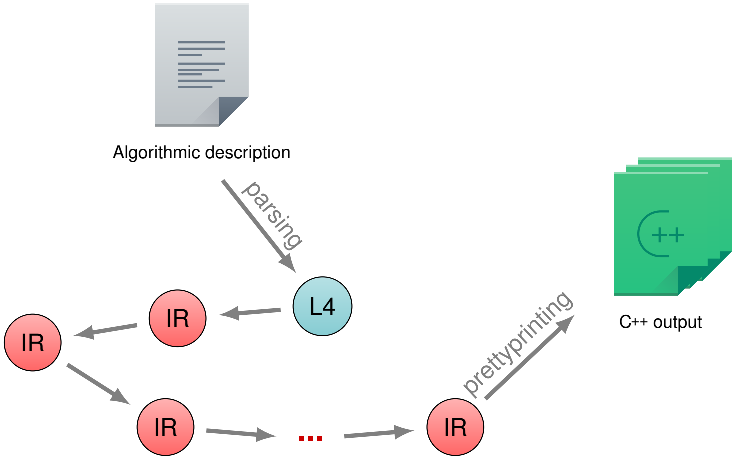
```
Function WCycle@(all, not(coarsest))() : Unit {
    repeat 4 times {
        Smoother@current()
    }
    UpResidual@current()
    Restriction@current()
    SetSolution@coarser(0)
    repeat 2 times {
        Wcycle@coarser()
    }
    Correction@current()
    repeat 3 times {
        Smoother@current()
    }
}

Function WCycle@coarsest() : Unit {
    /* ...direct solving... */
}
```



Transformation Framework

Transformation Framework



IR = mixture of ExaSlang 4 and C++

Transformation Framework

Current workflow

1. DSL input (Layer 4) is parsed
2. Parsed input is checked for errors and transformed into the IR
3. Many small, specialized transformations are applied
4. C++ output is prettyprinted

Transformation Framework

Current workflow

1. DSL input (Layer 4) is parsed
2. Parsed input is checked for errors and transformed into the IR
3. Many small, specialized transformations are applied
4. C++ output is prettyprinted

Concepts

- Major program modifications take place only in IR
- IR can be transformed to C++ code
- Small transformations can be enabled and arranged as necessary
- Central controller keeps track of program generation: StateManager
- Variant generation by program duplication at different transformation stages

Transformation Framework

Transformations

- Specify transitions between program states (abstract syntax trees)
- Are applied to program state in depth-first order
- May be applied to only part of the program state
- Are aggregated in strategies

Transformation Framework

Transformations

- Specify transitions between program states (abstract syntax trees)
- Are applied to program state in depth-first order
- May be applied to only part of the program state
- Are aggregated in strategies

Strategies

- Are applied in transactions
- Standard strategy executes all transformations in sequence
- Custom strategies possible

Transformation Framework

Transactions

- Before execution, a snapshot of the program state is taken
- May be committed or aborted

Transformation Framework

Transactions

- Before execution, a snapshot of the program state is taken
- May be committed or aborted

Checkpoints

- Copy of program state during compilation
- Serves to restore a program states
- Accelerates variant generation for design space exploration

Transformation Framework

Two example transformations:

```
var s = DefaultStrategy("example strategy")

// rename a certain stencil
s += Transformation("rename stencil", {
  case x : Stencil if(x.identifier == "foo")
    =>
    {
      if(x.entries.length != 7) error("invalid stencil size")
      x.identifier = "bar"; x
    }
})

// evaluate additions
s += Transformation("eval adds", {
  case AdditionExpression(l : IntegerConstant, r : IntegerConstant)
    => IntegerConstant(l.value + r.value)
})

s.apply // execute transformations sequentially
```

Optimizations

Polyhedral Model Extraction

Iteration domain

- Described by a single `loop over <field>`
- No need to deal with nested loops
- Consecutive loops may be merged

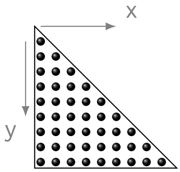
Polyhedral Model Extraction

Iteration domain

- Described by a single `loop over <field>`
- No need to deal with nested loops
- Consecutive loops may be merged

Memory accesses

Modelling takes place before accesses are linearized
➔ Allows to model code using, e.g., triangular fields




```
a[(y*y+y)/2+x]
```

vs.

```
a[y][x]
```

Polyhedral Model Optimization

Optimization steps

1. Compute dependences
2. Eliminate dead statement instances
3. Search an optimal schedule
   - choose from a number of custom schedulers
4. Tile individual dimensions in the model
   - tile shape is rectangular in the optimized target code
   - tile size can be prespecified, partially prespecified, or deduced
5. Rebuild abstract syntax tree

Polyhedral Model Optimization

Optimization steps

1. Compute dependences
2. Eliminate dead statement instances
3. Search an optimal schedule
   - choose from a number of custom schedulers, or
   - perform a complete search space exploration
4. Tile individual dimensions in the model
   - tile shape is rectangular in the optimized target code
   - tile size can be prespecified, partially prespecified, or deduced
5. Rebuild abstract syntax tree

Polyhedral Model Optimization

Optimization steps

1. Compute dependences
2. Eliminate dead statement instances
3. Search an optimal schedule
   - choose from a number of custom schedulers, or
   - perform a complete search space exploration
4. Tile individual dimensions in the model
   - tile shape is rectangular in the optimized target code
   - tile size can be prespecified, partially prespecified, or deduced
5. Rebuild abstract syntax tree

Optimization levels

0. Disable all polyhedral optimizations
1. Perform dependence analysis only
2. Optimize schedule, but do not tile
3. Do everything

Other Optimizations

- Address precalculation
  - Standard optimization in production compilers
  - Not always applied, since other transformations can stand in its way
  - ➔ We implemented a more advanced version directly

Other Optimizations

- Address precalculation
  - Standard optimization in production compilers
  - Not always applied, since other transformations can stand in its way
  - ➔ We implemented a more advanced version directly
- Arithmetic simplification
  - Convert division by a constant to multiplication by its inverse
  - Evaluate subexpressions as far as possible
  - Apply law of distributivity in order to
    - factor out repeated loads of the same array element
    - reduce the number of multiplications required

Other Optimizations

- Address precalculation
  - Standard optimization in production compilers
  - Not always applied, since other transformations can stand in its way
  - ➔ We implemented a more advanced version directly
- Arithmetic simplification
  - Convert division by a constant to multiplication by its inverse
  - Evaluate subexpressions as far as possible
  - Apply law of distributivity in order to
    - factor out repeated loads of the same array element
    - reduce the number of multiplications required
- Vectorization (SSE3, AVX, AVX2, QPX, NEON)
  - The use of vector units is mandatory to achieve best performance
  - Contemporary compilers are unable to emit efficient vector code
  - ➔ Explicit vectorization during generation

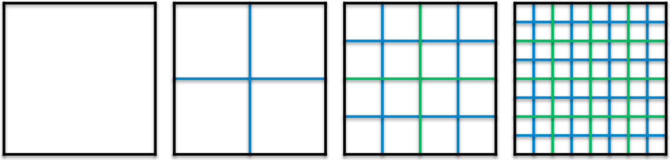
Other Optimizations

- Address precalculation
  - Standard optimization in production compilers
  - Not always applied, since other transformations can stand in its way
  - ➔ We implemented a more advanced version directly
- Arithmetic simplification
  - Convert division by a constant to multiplication by its inverse
  - Evaluate subexpressions as far as possible
  - Apply law of distributivity in order to
    - factor out repeated loads of the same array element
    - reduce the number of multiplications required
- Vectorization (SSE3, AVX, AVX2, QPX, NEON)
  - The use of vector units is mandatory to achieve best performance
  - Contemporary compilers are unable to emit efficient vector code
  - ➔ Explicit vectorization during generation
- Loop unrolling
  - Duplicate loop body to reduce branch penalty, condition evaluation, etc.
  - Two modes supported
    - Duplicate entire body at once
    - Duplicate each statement in-place
      (could be preferable on in-order architectures)
    - Special attention: reducing the number of loads in an interpolation

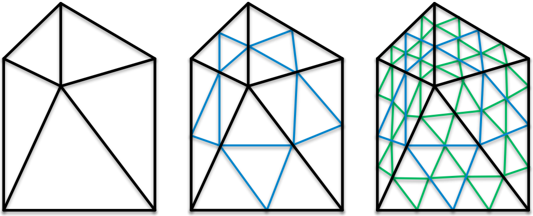
Partitioning the Computational Domain(s)

Domain Partitioning – Our Scope

- Presently: uniform grids



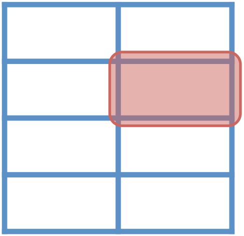
- Eventually: hierarchical hybrid grids (HHGs)



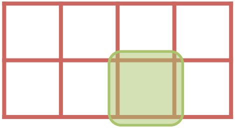
Domain Partitioning – Concept

- Easy for regular domains

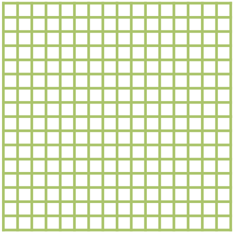
Each **domain** consists of one or more **blocks**



Each **block** consists of one or more **fragments**



Each **fragment** consists of several **data points / cells**



- More complicated for HHG

Domain Partitioning – Mapping to Parallelism

- Domain partition maps directly to different parallelization interfaces, e.g., MPI and OMP:
  - Each **block** corresponds to one *MPI* rank
  - Each **fragment** corresponds to one *OMP* rank
  - If single **fragment** per **block**, direct parallelization of kernels in *OMP*

Domain Partitioning – Mapping to Parallelism

- Domain partition maps directly to different parallelization interfaces, e.g., MPI and OMP:
  - Each **block** corresponds to one *MPI* rank
  - Each **fragment** corresponds to one *OMP* rank
  - If single **fragment** per **block**, direct parallelization of kernels in *OMP*
- Easily mapped to different interfaces:
  - PGAS
  - MPI and PGAS
  - MPI and CUDA

Domain Partitioning – Mapping to Parallelism

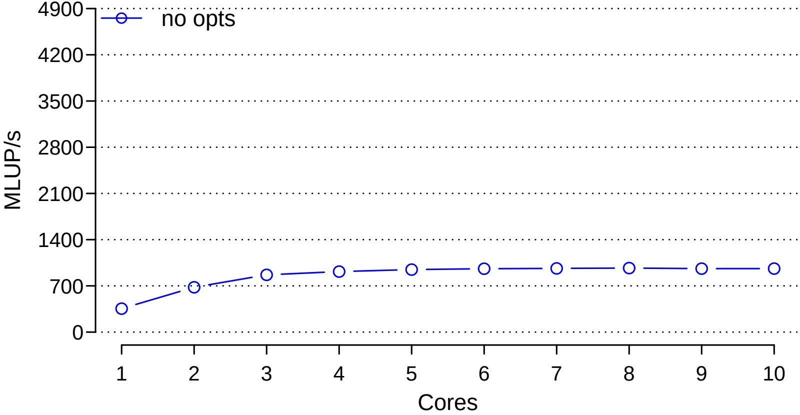
- Domain partition maps directly to different parallelization interfaces, e.g., MPI and OMP:
  - Each **block** corresponds to one *MPI* rank
  - Each **fragment** corresponds to one *OMP* rank
  - If single **fragment** per **block**, direct parallelization of kernels in *OMP*
- Easily mapped to different interfaces:
  - PGAS
  - MPI and PGAS
  - MPI and CUDA
- Communication – current state:
  - Mapping to fragments is declared at Layer 4
  - Communication statements are added automatically when transforming Layer 3 to Layer 4, where they may be reviewed or adapted
  - Actual realization, i.e., usage of synchronous and/or asynchronous MPI operations is up to the generator

Results

Node-Level Performance

3D 7-point Jacobi smoother

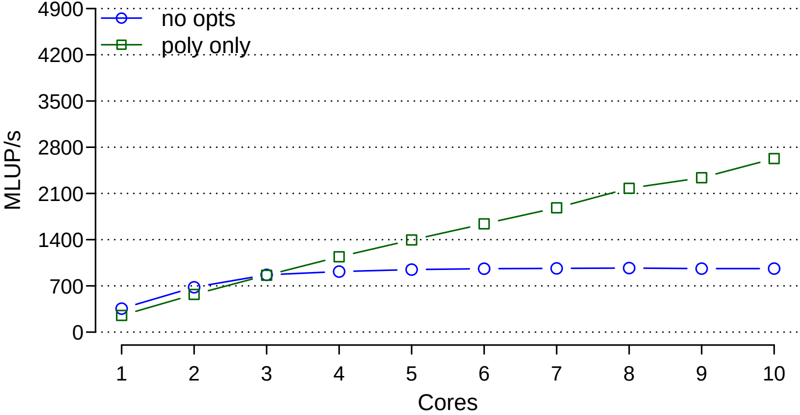
Intel IvyBridge EP



Node-Level Performance

3D 7-point Jacobi smoother

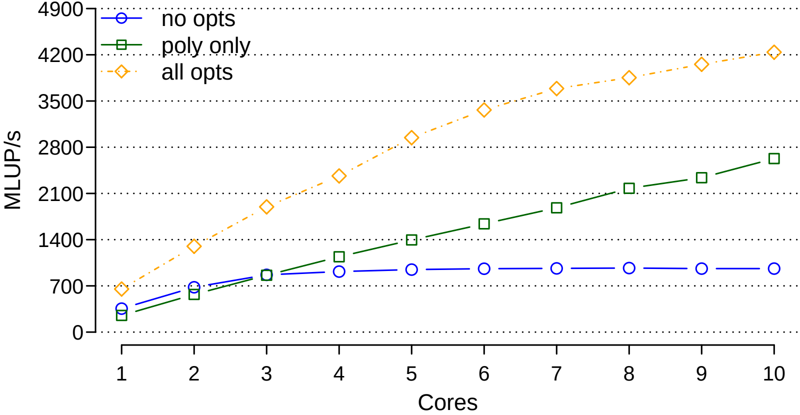
Intel IvyBridge EP



Node-Level Performance

3D 7-point Jacobi smoother

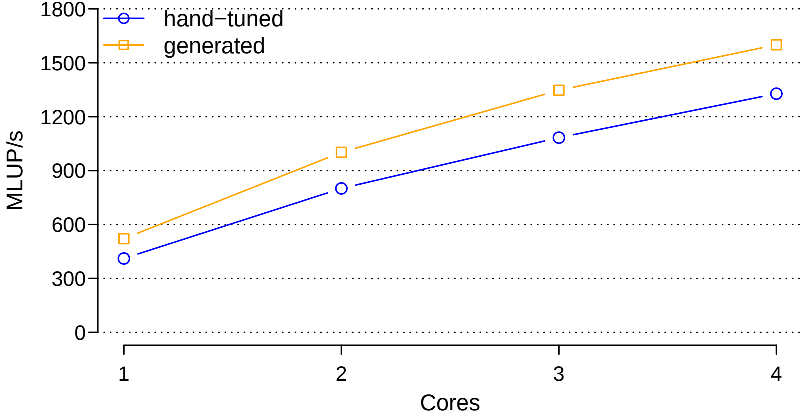
Intel IvyBridge EP



Node-Level Performance

3D 7-point Jacobi smoother

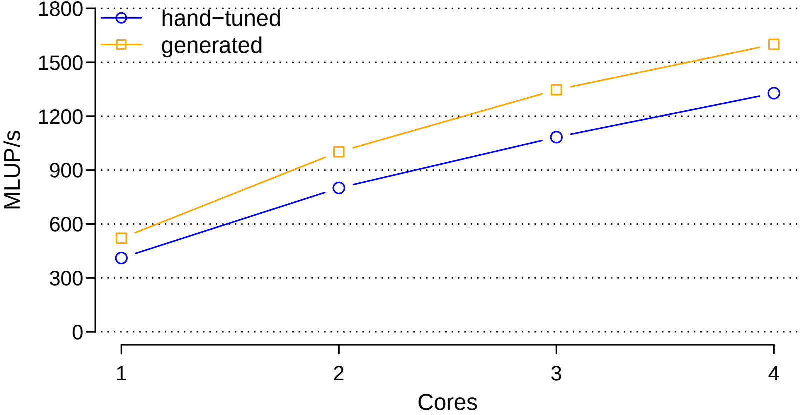
Intel IvyBridge (consumer)



Node-Level Performance

3D 7-point Jacobi smoother

Intel IvyBridge (consumer)



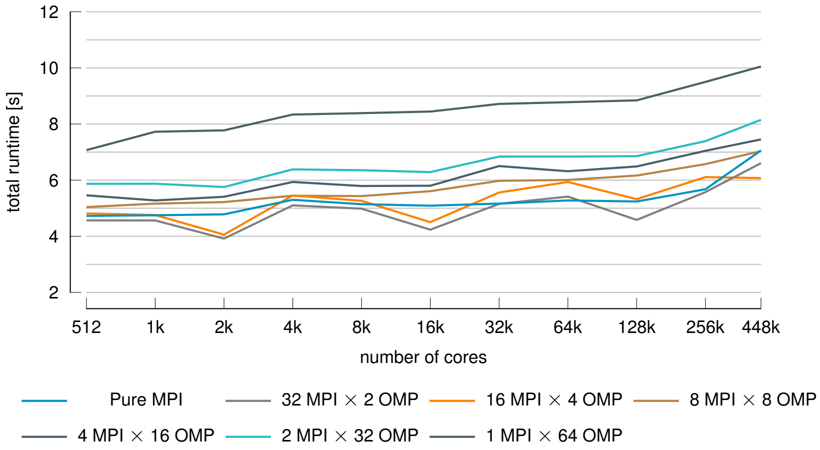
generated: 8 LOC ExaSlang 4 $\mapsto$ 4215 LOC C++ (up to 1108 characters per line)

Benchmark Problem and System

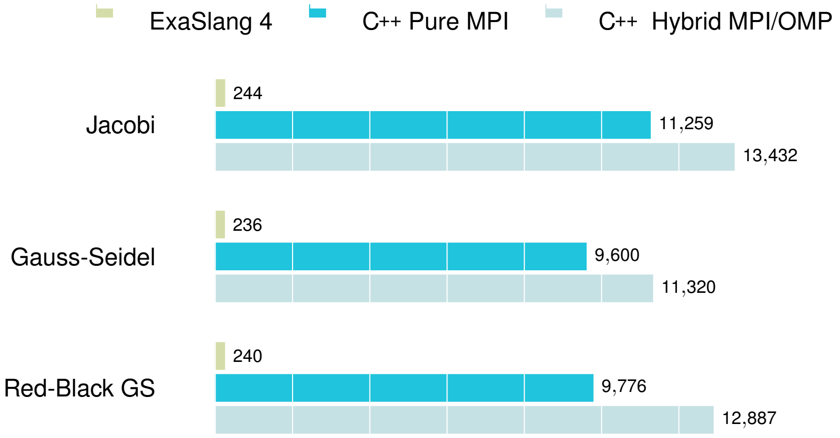
- Target system
  - JUQUEEN supercomputer located in Jülich, Germany
  - 458,752 cores / 28,672 nodes (1.6 GHz, 16 cores each, four-way multithreading)
- Considered problem
  - 3D finite differences discretization of Poisson's equation ($\Delta\phi = f$) with Dirichlet boundary conditions
  - V(3,3) cycle, parallel CG as direct solver (coarse grid solver)
  - Jacobi, Gauss-Seidel or red-black Gauss-Seidel smoother
  - pure MPI or hybrid MPI/OMP parallelization
  - 64 threads per node, roughly 10^6 unknowns per core
  - code optimized through polyhedral loop transformations, 2-way unrolling and address precalculation on finer levels as well as custom MPI data types
  - vectorization and blocking are not yet included

Weak Scalability

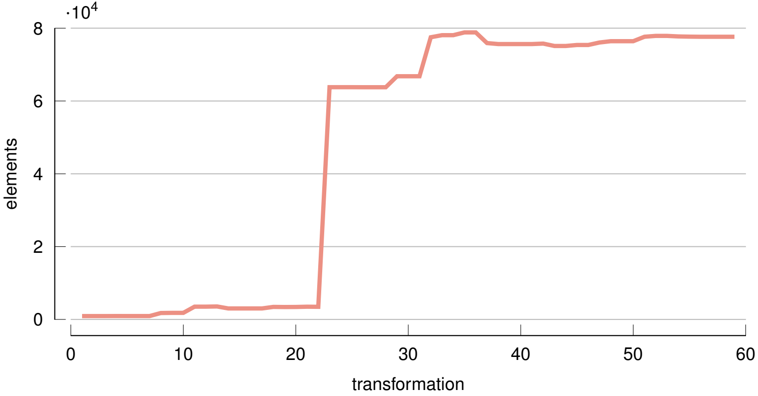
- Mean time per V-cycle
- V(3,3) with Jacobi and CG



ExaStencils Framework: Comparison of Lines of Code



ExaStencils Framework: Program Sizes during Transformation



ExaStencils Funding Period 2 (2016–2018)

Done and to be Done

	Done	To be Done
Domain	Scalar elliptic PDEs	Linear elasticity equations Navier-Stokes equations
Solvers	Jacobi Red-Black (Gauss-Seidel) Conjugate Gradient	Vanka (multiple unknowns) Multi-color smoothers Block smoothers
Clusters	BlueGene Q Intel	ARM
Accelerators	FPGAs	(multi-)GPUs
ExaSlang	Level 4	Levels 1–3
TPDL	Sketch	Full design
Applications		Non-Newtonian fluids Quantum chemistry Medical image processing
Performance	Execution speed	Energy awareness

Thanks for listening.

ExaStencils

ExaStencils – Advanced Stencil Code Engineering

http://www.exastencils.org

ExaStencils is funded by the German Research Foundation (DFG)
as part of the Priority Programme 1648 (Software for Exascale Computing).