Software design for highly scalable numerical algorithms

Harald Köstler
Workshop on Recent Advances in Parallel and High Performance Computing Techniques and Applications
12.1.2015, Singapore
Contents

- Introduction
- ExaStencils project goals
- Domain specific language design
- Numerical results
Introduction
Computational Science and Engineering

Applications
- Multiphysics
- fluid, structure
- medical imaging
- laser

Computer Science
- HPC / hardware
- Performance engineering
- software design

Applied Math
- LBM
- multigrid
- FEM
- numerics
Problems in Simulation

- **Multi-scale**: Many phenomena in physics and engineering can only be simulated when combining different models on different time and length scales.

- **Validation**: Often the assumptions in theoretical models do not match exactly to practical simulation results or no theoretical models are available for complex scenarios.
Problems in Applied Math

- **Discretization**: Choosing the „right“ discretization for a given physical model like finite differences, finite elements or discontinuous Galerkin is an art.

- **Solver/Optimization**: The variety of different mathematical models also results in a variety of methods to compute the quantity of interest.

- **Sensitivity**: Most realistic data in simulations comes with uncertainty, so the numerical methods must be analyzed with respect to their sensitivity in the input parameters.
Problems in High Performance Computing

- **Hardware:** Modern HPC platforms are massively parallel
  - Intra-core, intra-node, and inter-node
  - Heterogeneous architectures

- **Software:** CSE applications become more complex with increasing computational power
  - More complex models
  - Code development in interdisciplinary teams

- **Algorithm:** Class of different algorithms grows, many of them are just a general idea (like multigrid)
  - Components and parameters depend on grid, type of problem, …
Problems in Software Engineering

- **Performance**: Choose best hardware and algorithm
  - Time-consuming!
  - Language? Assembler?

- **Portability**: Prepare software for next generation hardware
  - Parallelization
  - Language for CPU/GPU/clusters? OpenCL? MPI?

- **Productivity**: Software should be usable, maintainable, extendable and it should be possible to develop it quickly
  - Documentation, Automated test environment
  - Language???
State of the Art: Application-driven Projects

User from application field $\rightarrow$ Description of application

Mathematician $\rightarrow$ Solution method

Software specialist $\rightarrow$ Parallel implementation and framework

Hardware specialist $\rightarrow$ Efficient implementation on specific hardware
Programming Approaches in CSE

● One problem – one code
  ● Everything is implemented from scratch or one uses common libraries
  ● Can be easily specialized and therefore optimized
  ● Problem has to be worth it

● One library – several problems
  ● Higher maintenance and user support effort
  ● Hard to fit all users needs and achieve optimal performance
  ● A whole community can benefit from it

● One generator – several problems
  ● Designing generator requires very high effort
  ● Problem-specific optimizations possible
Possible Generator Approach

- Choose a relatively narrow application domain
- Concentrate on few numerical methods
- Address the problems in exa-scale software development by automatic
  - code generation
  - code optimization techniques
  - co-design
Proposed: Domain-driven Projects

Users from different application fields

Description of application in domain specific language

Feature Model

Automatic selection of algorithmic components

Code generation for specific application

Automatic Tuning on specific hardware

Domain expert

Domain knowledge

PDE

\begin{align*}
\text{Operators::Laplacian(Data::solution) = Data::rhs}
\end{align*}

Mathematician

Software specialist

Hardware specialist
The ExaStencils Project
Project ExaStencils is funded by the German Research Foundation (DFG) as part of the Priority Program 1648 (Software for Exascale Computing) -> http://www.exastencils.org

- Sebastian Kuckuk
- Harald Köstler
- Ulrich Rüde

- Alexander Grebhahn
- Sven Apel

- Christian Schmitt
- Frank Hannig
- Jürgen Teich

- Hannah Rittich
- Matthias Bolten

- Stefan Kronawitter
- Armin Größlinger
- Christian Lengauer
Our approach to develop CSE software

- Start with an abstract problem formulation
  - A physical model
  - A numeric algorithm
  - A simple implementation
- Parse it into an abstract syntax tree using Scala
- Apply code transformations
  - Based on domain knowledge
  - Possible to add own ones
  - Can be highly non-local!
  - Add parallelization
- Enrich with external libraries
- Pretty-print to low-level code like C++/CUDA
Stencil – (Our) Definition

- A stencil describes pattern and non-zero entries of one row in a sparse matrix.

\[ A = \begin{bmatrix}
4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 & 0 \\
0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 & 0 \\
0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4
\end{bmatrix} \]
Stencil – (Our) Definition

- A stencil describes pattern and non-zero entries of one row in a sparse matrix.

\[ \nabla^2 F(x,y) = \frac{(F(x-1,y) + F(x+1,y) + F(x,y-1) + F(x,y+1) - 4F(x,y))}{h^2} \]
ExaStencils – Focus

- (elliptic) partial differential equations
- Geometric Multigrid solvers (GMG)
- Structured or semi-structured grids
- exa-scalable C++ or CUDA code
Nonlinear anisotropic diffusion

Requires to solve the time-dependent PDE

\[
div(g \nabla u) = \frac{\partial u}{\partial t} \quad \text{in } \Omega \times T
\]

\[
\langle g \nabla u, n \rangle = 0 \quad \text{on } \partial\Omega \times T
\]

\[
u(x,0) = u^0(x) \quad \text{in } \Omega
\]
Multigrid

- Goal: Solve a partial differential equation approximately on a discrete grid

$$\begin{align*}
\Omega \\
\Delta u &= f \quad \text{in } \Omega \\
u &= 0 \quad \text{in } \partial\Omega \\
A u_h &= f_h
\end{align*}$$

- An efficient method to solve such discretized PDEs in O(N) is multigrid
Multigrid

Smoothing of High Frequency Errors

Coarse Representation of Low Frequency Errors
Multigrid V-Cycle

- Smoothing
- Restriction
- Prolongation & Correction
- Coarse Grid Solving
Type of Grids

- Uniform grids

- Block-Structured grids
ExaStencils – Vision

- Generate exa-scalable C++ code for GMG solvers from
  - a high-level problem description specified by domain experts and
  - a target hardware architecture specification
- Provide different levels of abstraction that can be used as testing environments for
  - Mathematicians researching multigrid methods and components
  - Software Specialists researching programming languages, efficient communication strategies and program optimizations
  - Hardware Experts researching low-level and hardware-specific optimizations
Problem – Variance

- There is a lot of variance in the MG domain:
  - **Hardware**: CPU, GPU or both? Number of nodes, sockets and cores? Cache characteristics? Network characteristics?
  - **Software**: MPI, OpenMP or both? CUDA or OpenCL? Which version?
  - **MG components**: Cycle Type? Which smoother(s)? Which coarse grid solver? Which inter-Grid operators?
  - **MG parameters**: Relaxation? Number of smoothing steps? Other component dependent parameters?
  - **Optimizations**: Vectorization? (Software) Prefetching? Tiling? Temporal Blocking? Loop transformations?
  - **Problem description**: Which PDE? Which boundary conditions?
  - **Discretization**: Finite Differences, Finite Elements or Finite Volumes?
  - **Domain**: Uniform or block-structured? How to partition?
  - …
ExaStencils – Overview

- DSL as intuitive interface to the user
- Automatic deduction of configuration if desired
- Prediction and Optimization of the configuration’s performance using SPL and LFA
- Code generation in Scala
- Automatic hardware-specific optimizations
Aspects

- Code generation for specific HPC applications
- Domain-specific language design
- Co-design: Domain-specific knowledge representation and optimization
  - Hardware Description
  - Algorithmic Parameters
  - Transformation rules
- Efficient Algorithms
- Parallelization
- Performance Tuning
  - Polyhedral Optimization
  - Vectorization
ExaStencils – Layers

- Continuous Domain & Continuous Model
  - discretization
- Discrete Domain & Discrete Model
  - parametrization
- Algorithmic Components & Parameters
  - specification
- Complete Program Specification
  - optimization
- Internal Representation
  - generation
- Exascale C++ Code
DSL overview
DSL Scheme

Layer 1
- Computational Domain
- Continuous Model

Layer 2
- Discrete Domain
- Discrete Model

Layer 3
- Algorithmic Parameters and Components
- Application Settings

Layer 4
- Pseudo Code
Problem Description (continuous)

**Domain** \( \omega = [0,1] \times [0,2] \)

\( f : \omega \to \mathbb{R}^1 \)
\( u : \omega \to \mathbb{R}^1 \)

Laplacian : ( \( \omega \to \mathbb{R}^1 \) ) \( \to \) ( \( \omega \to \mathbb{R}^1 \) )

Laplacian = \( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \)

**pde** : Laplacian \( [ u ] = f \) in \( \omega \)

**bc** : \( u = 0 \) in partial_\( \omega \)
Generation of discrete problem

- Domain-specific knowledge
  - Discretization methods FD, FV, FE
  - Types of operators supported that result in sparse matrices
- Domain-specific optimization chooses type of discretization and e.g. concrete data types
- Description is parsed, an abstract syntax tree is constructed and then transformed into a discrete representation of the problem
- Code generation framework is implemented in Scala language
Problem Description (discrete)

**Fragments** \( f_1 = \text{Regular} \_\text{Square} \)

**Discrete\_Domain** \( \omega \) levels 10 { 
\[
\begin{align*}
\text{xsize}[0] &= 1024 \\
\text{ysize}[0] &= 1024 \\
\text{xsize}[l+1] &= \text{xsize}[l] / 2 \\
\text{ysize}[l+1] &= \text{ysize}[l] / 2 
\end{align*}
\]
}

**Field**<Double>@nodes \( f \)  
**Field**<Double>@nodes \( u \)  
**StencilMatrix**<Double,FD,2>@nodes Laplacian
Multigrid Algorithm

Algorithm 1 Recursive V-cycle: $u_{h}^{(k+1)} = V_h(u_{h}^{(k)}, A^h, f^h, v_1, v_2)$

1: if coarsest level then
2: solve $A^h u^h = f^h$ by a (parallel) direct solver or by CG iterations
3: else
4: $\tilde{u}_{h}^{(k)} = S_v^1(u_{h}^{(k)}, A^h, f^h)$ \{presmoothing\}
5: $r^h = f^h - A^h \tilde{u}_{h}^{(k)}$ \{compute residual\}
6: $r^{H} = R r^{h}$ \{restrict residual\}
7: $e^{H} = V_H(0, A^{H}, r^{H}, v_1, v_2)$ \{recursion\}
8: $\tilde{u}_{h}^{(k)} = \tilde{u}_{h}^{(k)} + P e^{H}$ \{prolongate error and do coarse grid correction\}
9: $u_{h}^{(k+1)} = S_v^2(\tilde{u}_{h}^{(k)}, A^h, f^h)$ \{postsmoothing\}
10: end if
And now simply transform high-level Matlab-like code to a parallel low-level C++ or CUDA implementation …
## Runtime Results for Different Problems [Koestler14]

Sizes 4096x4096 resp. 256x256x256

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cpu</td>
<td></td>
<td>gpu</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>8</td>
<td>256</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Double</td>
<td>Float</td>
<td>Double</td>
<td>Float</td>
<td>Double</td>
<td>Float</td>
</tr>
<tr>
<td>Laplace</td>
<td>2D</td>
<td>Jacobi</td>
<td>546</td>
<td>1.17</td>
<td>3.50</td>
<td>3.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GaussSeidel</td>
<td>453</td>
<td>1.16</td>
<td>2.42</td>
<td>3.24</td>
</tr>
<tr>
<td></td>
<td>3D</td>
<td>Jacobi</td>
<td>608</td>
<td>1.08</td>
<td>2.60</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GaussSeidel</td>
<td>608</td>
<td>1.08</td>
<td>2.60</td>
<td>3.01</td>
</tr>
<tr>
<td>Complex Diffusion</td>
<td>2D</td>
<td>Jacobi</td>
<td>9235</td>
<td>1.02</td>
<td>2.81</td>
<td>2.87</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GaussSeidel</td>
<td>7504</td>
<td>1.11</td>
<td>2.18</td>
<td>2.29</td>
</tr>
<tr>
<td></td>
<td>3D</td>
<td>Jacobi</td>
<td>8799</td>
<td>1.15</td>
<td>2.77</td>
<td>2.91</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GaussSeidel</td>
<td>9048</td>
<td>1.80</td>
<td>2.60</td>
<td>2.87</td>
</tr>
</tbody>
</table>

ms speedup speedup speedup speedup speedup speedup
DSL layer 4
Problem I

- We want to compare different smoothers …
  - … with their respective best performance and that means
  - … with (automatically) inlined stencil convolutions
  - … with different optimizations for each MG level
- We don’t want …
  - … copy-paste every function for each case

```c
void smoother_GS_1();
void smoother_GS_2();
void smoother_GS_3();
void smoother_GS_4();
...
void smoother_Jac_1();
void smoother_Jac_2();
void smoother_Jac_3();
void smoother_Jac_4();
...
void smoother_RBGS_1();
void smoother_RBGS_2();
void smoother_RBGS_3();
void smoother_RBGS_4();
...
```
Leveled Structures in Layer 4

- Layer 4 provides level concept for relevant structures

  Field Residual< ... >@all
  Stencil LaplStencil@all { ... }
  StencilField Lapl< ... >@all { ... }
  def Smoother@((coarsest + 1) to finest) ( ) : Unit { ... }

- A separate version of the given structure is generated for each level
- Levels can be numbers or special keywords (e.g. all, finest, coarsest)
Layer 4 Functions – Example

**function** VCycle@coarsest ( ) : Unit { /* coarse grid solver */ }

**function** VCycle@((coarsest + 1) to finest) ( ) : Unit {
  repeat up 1 {
    Smoother@current ( )
  }
  UpResidual@current ( )
  Restriction@current ( )
  SetSolution@coarser ( 0 )
  VCycle@coarser ( )
  Correction@current ( )
  repeat up 2 {
    Smoother@current ( )
  }
}

Problem II

- We want to have data fields on a specific subset of MG levels with varying …
  - … numbers of points
  - … numbers of ghost layers
  - … communication behavior
  - … padding

- We don’t want to …
  - … trade variability for speed
  - … re-implement every possible scenario
Fields

- Every field is specified on layer 4
- Requires identifier, data type, domain, layout and BC

Field Solution< Real, global, BasicComm, 0.0 >[2]@ (coarsest to (finest - 1))
Field Solution<
   Real, global, BasicComm, ( sin ( M_PI * xPos ) * … ) >[2]@ finest
Field Residual< Real, global, BasicComm, None >@ all
Field RHS< Real, global, NoComm, None >@ all

- Fields can …
  - … be slotted which corresponds to an array of identical fields
  - … can have array data types, i.e.
  - … be scalar, vector or coefficient fields
Field Layouts

- Specifies how much memory the field is using and how this memory is assigned
- Used, among other things, to generate communication functions

```
Layout BasicComm {
    ghostLayers = [ 1, 1 ] with communication
    duplicateLayers = [ 1, 1 ] with communication
    // innerPoints = [] -> defaulted to 2^l + 1 - 2 * dup
}

Layout NoComm {
    ghostLayers = [ 0, 0 ]
    duplicateLayers = [ 1, 1 ]
    // innerPoints = [] -> defaulted to 2^l + 1 - 2 * dup
}```
Fields – Resulting Generator Classes (IR level)

```scala
class FieldLayoutPerDim(
    var numPadLayersLeft : Int,
    var numGhostLayersLeft : Int,
    var numDupLayersLeft : Int,
    var numInnerLayers : Int,
    var numDupLayersRight : Int,
    var numGhostLayersRight : Int,
    var numPadLayersRight : Int)

case class Field(
    var identifier : String,
    var index : Int,
    var domain : Domain,
    var dataType : Datatype,
    var layout : Array[FieldLayoutPerDim],
    var communicatesDuplicated : Boolean,
    var communicatesGhosts : Boolean,
    var level : Int,
    var numSlots : Int,
    var referenceOffset : MultiIndex,
    var dirichletBC : Option[Expression])

case class FieldSelection(
    var field : Field,
    var slot : Expression,
    var arrayIndex : Int,
    var fragIdx : Expression = LoopOverFragments.deflt)
```
Problem III

- We want to have stencils with …
  - … constant coefficients
  - … varying but location-independent coefficients
  - … different coefficients for each grid point

- We don’t want to …
  - … indirect my stencil accesses if not required
  - … re-implement functions for different stencils
  - … be restricted to a specific stencil shape
Stencils and StencilFields

- Stencils can either be fixed

  Stencil LaplStencil@all {
  [ 0, 0] => 4
  [ 1, 0] => -1
  [-1, 0] => -1
  [ 0, 1] => -1
  [ 0, -1] => -1
  }

- Or variable

  Stencil LaplStencil@all { ... }  Field LaplCoeff< Array[Real][5], global, NoComm, None >@all
  StencilField Lapl< LaplCoeff => LaplStencil >@all
Stencil Convolutions

- Stencil convolutions, i.e. multiplications between
  - Stencils and fields
  - StencilFields and fields
  are automatically detected (and resolved in a separate step)

object FindStencilConvolutions extends DefaultStrategy("FindStencilConvolutions") {
  this += new Transformation("SearchAndMark", {
    case MultiplicationExpression(StencilAccess(stencil),
      fieldAccess : FieldAccess) =>
      StencilConvolution(stencil, fieldAccess)
    case MultiplicationExpression(stencilFieldAccess : StencilFieldAccess,
      fieldAccess : FieldAccess) =>
      StencilFieldConvolution(stencilFieldAccess, fieldAccess)
  })
}
Example: Jacobi Smoother

```plaintext
function Smoother@((coarsest + 1) to finest) ( ) : Unit {
  communicate Solution[0]@(current)
  loop over inner on Solution@(current) {
    Solution[1]@current =
    Solution[0]@current
    + (( ( 1.0 / diag ( Lapl@(current) ) ) * 0.8 )
    * ( RHS@current - Lapl@(current) * Solution[0]@current ) )
  }
  ...
```
Jacobi Smoother – Resulting Code (w/o basic Opt)

```c
#include "MultiGrid/MultiGrid.h"

void Smoother_4() {
    exchsolutionData_4(0);
    #pragma omp parallel for schedule(static) num_threads(8)
    for (int fragmentIdx = 0; fragmentIdx < 8; ++fragmentIdx) {
        if (isValidForSubdomain[fragmentIdx][0]) {
            for (int y = iterationOffsetBegin[fragmentIdx][0][1];
                 y < (iterationOffsetEnd[fragmentIdx][0][1]+17); y += 1)
                for (int x = iterationOffsetBegin[fragmentIdx][0][0];
                     x < (iterationOffsetEnd[fragmentIdx][0][0]+17); x += 1)
                    slottedFieldData_Solution[1][fragmentIdx][4][((y)*19)+19+(x+1)] =
                        (slottedFieldData_Solution[0][fragmentIdx][4][((y)*19)+19+(x+1)])
                          +(((1.0e+00/fieldData_LaplCoeff[fragmentIdx][4][((y)*17)+x])*8.0e-01)
                           *(fieldData_RHS[fragmentIdx][4][((y)*17)+x])
                          -((((fieldData_LaplCoeff[fragmentIdx][4][((y)*17)+x])
                          *slottedFieldData_Solution[0][fragmentIdx][4][((y)*19)+19]+(x+1)))+
                          (fieldData_LaplCoeff[fragmentIdx][4][(((y)*17)+289)+x])
                          *slottedFieldData_Solution[0][fragmentIdx][4][((y)*19)+19]+(x+2)))+
                          (fieldData_LaplCoeff[fragmentIdx][4][(((y)*17)+578)+x])
                          *slottedFieldData_Solution[0][fragmentIdx][4][((y)*19)+19]+x))
                          +(fieldData_LaplCoeff[fragmentIdx][4][(((y)*17)+867)+x])
                          *slottedFieldData_Solution[0][fragmentIdx][4][((y)*19]+38+(x+1)))+
                          ((fieldData_LaplCoeff[fragmentIdx][4][(((y)*17)+1156)+x])
                          *slottedFieldData_Solution[0][fragmentIdx][4][((y)*19)+(x+1)]));
        }
    }
}
```

#include "MultiGrid/MultiGrid.h"

void Smoother_4() {
    #pragma omp parallel for schedule(static) num_threads(8)
    for (int fragmentIdx = 0; fragmentIdx < 8; ++fragmentIdx) {
        if (isValidForSubdomain[fragmentIdx][0]) {
            for (int c0 = iterationOffsetBegin[fragmentIdx][0][1];
                c0<=(iterationOffsetEnd[fragmentIdx][0][1]+16); c0 = (c0+1))
                double* slottedFieldData_Solution_1_fragmentIdx_4_p1 =
                    &(slottedFieldData_Solution[1][fragmentIdx][4][(19*c0)]);
                double* fieldData_RHS_fragmentIdx_4_p1 = &(fieldData_RHS[fragmentIdx][4][(17*c0)]);
                double* slottedFieldData_Solution_0_fragmentIdx_4_p1 = ...
                double* fieldData_LaplCoeff_fragmentIdx_4_p1 = ...
            for (int c1 = iterationOffsetBegin[fragmentIdx][0][0];
                c1<=(iterationOffsetEnd[fragmentIdx][0][0]+16); c1 = (c1+1)) {
                slottedFieldData_Solution_1_fragmentIdx_4_p1[(c1+20)]=
                    (slottedFieldData_Solution_0_fragmentIdx_4_p1[(c1+20)]
                        + (((1.0e+00/fieldData_LaplCoeff_fragmentIdx_4_p1[c1])*8.0e-01)*(fieldData_RHS_fragmentIdx_4_p1[c1])
                        - (((fieldData_LaplCoeff_fragmentIdx_4_p1[c1]*slottedFieldData_Solution_0_fragmentIdx_4_p1[(c1+20)])
                            + (fieldData_LaplCoeff_fragmentIdx_4_p1[c1+289]*slottedFieldData_Solution_0_fragmentIdx_4_p1[(c1+21)]))))
                        + (fieldData_LaplCoeff_fragmentIdx_4_p1[(c1+578)]*slottedFieldData_Solution_0_fragmentIdx_4_p1[(c1+19)]))
                        + (fieldData_LaplCoeff_fragmentIdx_4_p1[(c1+867)]*slottedFieldData_Solution_0_fragmentIdx_4_p1[(c1+39)]))
                        + (fieldData_LaplCoeff_fragmentIdx_4_p1[(c1+1156)]*slottedFieldData_Solution_0_fragmentIdx_4_p1[(c1+1)]));
                ...
            }
Parallelization
Problem IV

● We want my program to run …
  ● … in serial
  ● … in parallel with MPI and/ or OMP

● We want to …
  ● … dynamically set up all required neighborhood information (if the domain allows it)
  ● … add/ remove MPI/ OMP code segments

● Again, we don’t want to …
  ● … copy-paste lots of code
Domain Partitioning

- 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

- Two different ways:
  - Load domain information from file
  - If supported by domain specification: generate req. information on the fly
- All domains are specified in layer 4
  
  Domain global\(< [ -1, -1 ] \) to \([ 1, 1 ] \)>
  
  Domain sthSmaller\(< [ -0.75, -0.75 ] \) to \([ 0.75, 0.75 ] \)>

- Actual partition will be determined by SPL or overwritten by the user
  
  dimensionality = 2

  domain\_numBlocks\_x = 2
  domain\_numBlocks\_y = 4

  domain\_numFrgs\_PerBlock\_x = 4
  domain\_numFrgs\_PerBlock\_y = 2
Domain Partitioning

- Domain partition maps directly to parallelization strategy:
  - Each **block** corresponds to one **MPI** rank
  - Each **fragment** corresponds to one **OMP** rank
  - Pure **MPI** corresponds to one **fragment/block**
  - Pure **OMP** corresponds to one **block**
  - Hybrid **MPI/OMP** corresponds to multiple **blocks** and multiple **fragments/block**
  - Alternative approach: aggregate all **fragments** within one **block** and **OMP** parallelize field operations directly
Domain Partitioning - Resulting Code

#include "Domains/DomainGenerated.h"

void initDomain() {
  Vec3 positions[8];
  unsigned int posWritePos = 0;
  Vec3 rankPos(mpiRank % 2, (mpiRank / 2) % 4, mpiRank / 8);

  for (int y = 0; y < 2; y += 1)
    for (int x = 0; x < 4; x += 1)
      positions[posWritePos++] = (
        Vec3(((rankPos.x*4)*2.5e-01)+1.25e-01)+(x*2.5e-01)*-1.0e+00,(((rankPos.y*2)*2.5e-01)+1.25e-01)+(y*2.5e-01)1.0e+00,0));

  for (int fragmentIdx = 0; fragmentIdx < 8; ++fragmentIdx) {
    someRelevantInfo[fragmentIdx] = …
  }

#pragma omp parallel for schedule(static) num_threads(8)
  for (int fragmentIdx = 0; fragmentIdx < 8; ++fragmentIdx) {
    isValidateForSubdomain[fragmentIdx][0] = (((pos[fragmentIdx].x>=-1.0e+00)&(pos[fragmentIdx].x<=1.0e+00))&((pos[fragmentIdx].y>=-1.0e+00))&((pos[fragmentIdx].y<=1.0e+00)));
    …

    Vec3 offsetPos = (pos[fragmentIdx]+Vec3(-1*2.5e-01, 0 * 2.5e-01, 0 * 0.0));
    if (isValidForSubdomain[fragmentIdx][0] & ((((offsetPos.x>=-1.0e+00)&(offsetPos.x<=1.0e+00))&((offsetPos.y>=-1.0e+00)) & (offsetPos.y<=1.0e+00)))
      (mpiRank ==(((offsetPos.x<-1.0e+00)||(offsetPos.x>1.0e+00)||(offsetPos.y<-1.0e+00)||(offsetPos.y>1.0e+00))))
        (MPI_PROC_NULL) : 
          (((int)floor(((offsetPos.y+1.0e+00)/2.5e-01))/2)*2)+(int)floor(((offsetPos.x+1.0e+00)/2.5e-01))/4))} 
      connectLocalElement(fragmentIdx, ((((int)floor(((offsetPos.y+1.0e+00)/2.5e-01))/2)*2)+(int)floor(((offsetPos.x+1.0e+00)/2.5e-01)/4)), 0, 0); 
    else 
      connectRemoteElement(fragmentIdx, ((((int)floor(((offsetPos.y+1.0e+00)/2.5e-01))/2)*2)+(int)floor(((offsetPos.x+1.0e+00)/2.5e-01))/4), 
        ((((offsetPos.x<-1.0e+00)||(offsetPos.x>1.0e+00)||(offsetPos.y<-1.0e+00)||(offsetPos.y>1.0e+00)))
          (MPI_PROC_NULL) : 
            (((int)floor(((offsetPos.y+1.0e+00)/2.5e-01))/2)*2)+(int)floor(((offsetPos.x+1.0e+00)/2.5e-01)/4)), 0, 0); 
  }

  …
Problem V

- We need to exchange data between processes ...
  - ... locally and/or remotely
  - ... with different patterns (e.g. 6P/26P in 3D)
  - ... asynchronously
  - ... using MPI data types if reasonable
  - ... for each possible field layout

- We don’t want to ...
  - ... implement every possible case
  - ... extensively use templates and defines
  - ... trade variability for performance (e.g. using PGAS)
The 4/6 case requires a larger data volume to be transferred as well as separate and synchronized updates in x, y and z direction.
Communication – User Interface

- Communication statements are added automatically or specified by the user
  
  /* communicates all applicable layers */
  communicate Solution@current
  /* communicates only ghost layers */
  communicate ghost of Solution[curSlot]@current
  /* communicates duplicate and first two ghost layers */
  communicate dup ghost [0, 1] of Solution[curSlot]@current
  /* starts asynchronous communicate */
  begin communicate Residual@current

- Basic (Layer 4) communicate statements are synchronous with respect to the computations
- Actual realization, i.e. usage of synchronous and/ or asynchronous MPI operations is up to the generator
Communication – Node Progression

communicate Solution@current

ExchangeDataFunction

FunctionCall

RemoteSends

LocalSends

RemoteRecvs

LoopOverFragments

ConditionStatement(iv.IsValidForSubdomain(...))

CopyToSendBuffer

RemoteSend

WaitForTransfer

CopyFromRecvBuffer

RemoteRecv

WaitForTransfer

LoopOverDimensions

Direct Copy

for duplicate and/or ghost

for each neighbor
## Lines of Code for typical configurations

<table>
<thead>
<tr>
<th>Smoother</th>
<th>Stencil</th>
<th>Comm.</th>
<th>LOC DSL</th>
<th>LOC C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>Constant</td>
<td>Simple</td>
<td>231</td>
<td>12,821</td>
</tr>
<tr>
<td>Jacobi</td>
<td>Constant</td>
<td>Extended</td>
<td>231</td>
<td>28,331</td>
</tr>
<tr>
<td>Jacobi</td>
<td>Varying</td>
<td>Simple</td>
<td>292</td>
<td>19,330</td>
</tr>
<tr>
<td>Jacobi</td>
<td>Varying</td>
<td>Extended</td>
<td>292</td>
<td>47,814</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>Constant</td>
<td>Simple</td>
<td>227</td>
<td>12,726</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>Constant</td>
<td>Extended</td>
<td>227</td>
<td>28,230</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>Varying</td>
<td>Simple</td>
<td>288</td>
<td>19,229</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>Varying</td>
<td>Extended</td>
<td>288</td>
<td>47,713</td>
</tr>
</tbody>
</table>
Numerical Results
IBM BlueGene/Q JUQUEEN in Jülich

- 28,672 Nodes (458,752 Cores)
- Compute Node: IBM PowerPC A2, 1.6 GHz, 16 cores
- Main memory: 16 GB per node (aggregate 448 TB)
- Overall peak performance: 5.9 PetaFLOP/s
Basic Weak Scaling: Laplace Problem

Weak Scaling for two Configurations

Number of Cores

Mean Time per vCycle [ms]

V3,3 with Gauss-Seidel

V4,2 with Jacobi
Basic Weak Scaling: Laplace Problem

Weak Scaling for two Configurations

- V3,3 with Gauss-Seidel
- V4,2 with Jacobi

Number of Cores

- 512, 1k, 2k, 4k, 8k, 16k, 32k, 64k, 128k, 256k

Parallel Efficiency

- 0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2
Application: Stochastic variable heat conduction [Loh14]

\[ \nabla \cdot (e^{a(x)} \nabla U(x)) = 0 \]

\[ U(0, x_2) = 3 \]
\[ U(1, x_2) = 5 \]
\[ U(x_1, 0) = 10 \]
\[ U(x_1, 1) = 1 \]

\[ x \in \Omega = [0, 1] \times [0, 1] \]
\[ x_2 \in \partial \Omega_W = [0, 1] \]
\[ x_2 \in \partial \Omega_E = [0, 1] \]
\[ x_1 \in \partial \Omega_S = [0, 1] \]
\[ x_1 \in \partial \Omega_N = [0, 1] \]
Gaussian field approximation

\[ \text{Cov}(a(x_1), a(x_2)) = \frac{\sigma^2}{2^{\nu-1} \Gamma(\nu)} (\kappa \|x_1 - x_2\|)^\nu K_\nu(\kappa \|x_1 - x_2\|) \]

\[ (\kappa^2 - \Delta)^{\alpha/2}(\tau a(x)) = W(x) \]

where \( x, x_1, x_2 \in \Omega^d, \ W(x) \sim \text{MVN}(0, 1) \)

\[ \nabla a(x) \cdot \hat{n} = 0 \quad x \in \partial \Omega \]

\[ \tau^2 = \frac{\Gamma'(\nu)}{\Gamma(\nu)(4\pi)^{d/2} \kappa^{2\nu} \sigma^2} \]

\[ \kappa = \frac{\sqrt{8\nu}}{\lambda} \]

\[ \nu = \alpha - \frac{d}{2} \]
Gaussian Markov random field for single realization

(a) GMRF
(b) Solution

\[ \lambda = 0.1, \sigma = 1 \]
Expectation of solution field (100 samples, Monte Carlo simulation)
Conclusion

- When writing traditional code often different components with very similar structure occur, i.e. parts of code that vary only in one small detail
- Traditional approaches are
  - Modularizing, i.e. wrapping in classes, functions, etc.
  - Duplicating parts of the code by copy-pasting, templating, defines, etc.
- The main problems usually are granularity, code quality and performance
- In consequence, a huge number of configurations is usually not efficiently programmable in one big framework (partly due to diverse dependencies between different parts of the code)
- In our opinion, code generation is a viable alternative that can alleviate some of the common issues
References (full list on http://www.exastencils.org/)


Thank you for your Attention!

Questions?