OPM DEVELOPMENT STORIES

Atgeirr Flø Rasmussen
About SINTEF

Vision: technology for a better society

• independent, not-for-profit organization
• largest for-contract research in Scandinavia, fourth largest in Europe
• 2100 employees
• NOK 3.1 billion turnover, 90% ‘won’ in open competition
• more than 7000 research projects for some 2300 clients
• offices in Trondheim, Oslo, Bergen, Brussels, Houston, . . .
Computational Geosciences group

• One of eight research groups at the department of Mathematics & Cybernetics, SINTEF Digital
• 14 researchers/postdocs/PhD students
• Offices in Oslo, Norway
• Performs a mixture of basic and applied research
• Well known for our open-source software: MRST and OPM
• Internationally oriented
• Strong publication record
• Main clients: Equinor, ExxonMobil, Research Council of Norway, Wintershall, . . .
Some OPM development stories

• Grid interfaces: a personal history
• Making Flow perform well
Grid interfaces: a personal history
Computational grids discretize the spatial domain.

Broad topology categories:
- Structured/Cartesian
- Block-structured
- Unstructured/tetrahedral
- Fully unstructured
Reservoir grids are "bad"

- Fully unstructured (arbitrary connectivity)
- Bad cell shapes
- Huge anisotropy ratios
- Very heterogeneous properties
Important grid concepts

**Topology:**

- Cells, Faces, Edges, Vertices
  - *Entities* that make up the grid, *dimension* 3, 2, 1, 0.
  - Form a *cell complex*: (n-1)-dim entities are intersections of n-dim entities
  - Every entity is *oriented* (not always used)

- Adjacency relations
  - Which entities (of different dimension) are adjacent
  - Example: for face $f_0$, its neighbour edges are $\{-e_0, e_1, -e_2\}$

**Geometry:**

- Positions, volumes, areas, normals, centroids...
  - Anything depending on the embedding in $\mathbb{R}^3$
Some grid interfaces I have used (1)

1. Interface used in our first upscaling codes (2006-2007)
   • A single class representing a fully unstructured grid
   • No templates
   • Grid assumed to be unchanging after construction
   • Example methods:
     
     ```c++
     void getCellsForFace(Index f, Index& c1, Index& c2)
     bool halfFaceIsFace(Index half_face_ix, Index face_ix) const
     void halfFacesOfFace(Index face_ix, Index& hface1_ix, Index& hface2_ix) const
     void coord(Index corner, FloatType* p) const
     ```
   
   • Main benefits: straightforward, random access.
2. Interface used in our inhouse simulation codes (2007-2010)
   • Several grid class variants: Cartesian, Cornerpoint, Tetrahedral
   • Heavily templated! Very exotic to some.
   • Grid assumed to be unchanging after construction
   • Example code (adjacency relation: cells adjacent to a face):
     
     ```cpp
     typename grid_t::range_t cells_of_face
     = grid.template neighbours<FaceType, CellType>(face_index);
     for (int cell_index : cells_of_face) {
       // Do something with cell_index
     }
     ```
   • Main benefits: arbitrary topological relations, random access
Some grid interfaces I have used (3)

3. The Dune grid interface (2009-today)
   • Many grid class variants. CpGrid provides corner-point grids
   • Heavily templated!
   • Limited adjacency relations (but has cell->face and face->cell)

```cpp
elemIt = gridView.template begin</*codim=*/ 0>();
for (; elemIt != elemEndIt; ++elemIt) {
    auto isIt = gridView.ibegin(*elemIt);
    const auto& isEndIt = gridView.iend(*elemIt);
    for (; isIt != isEndIt; ++isIt) {
        if (isIt->boundary()) {
            // deal with grid boundaries
        }
    }
}
```

• Main benefits: suitable for parallel and adaptive use, open source and not inhouse
4. The UnstructuredGrid struct (2009-2015[ish])

- A simple C struct, similar to grid structure in MRST
- Access topological and geometric information as simple arrays
- CRS-like structure for unstructured adjacency relations

```c
for (int i = grid.cell_facepos[cell]; i < grid.cell_facepos[cell+1]; ++i) {
    int f = grid.cell_faces[i];
    if (cell == grid.face_cells[2*f]) {
        flux = darcyflux[f];
    } else {
        flux = -darcyflux[f];
    }
}
```

- Main benefits: easy to integrate with MRST, simple (although some conventions used might surprise you)
5. The free function grid interface in OPM (2014-today)

- All based on free functions taking a grid as one function argument
- Overloaded on grid class to support CpGrid and UnstructuredGrid

```cpp
auto c2f = cell2Faces(grid);
for(auto it = c2f.begin(), end = c2f.end(); it != end; ++it) {
    const int face_index = *it;
    const double area = faceArea(grid, face_index);
}
```

- Main benefits: allowed code to target both UnstructuredGrid and CpGrid (dune interface)
Even more grid interfaces I have used...

6. A grid used in an old corner-point GRDECL preprocessor (2006-2009)

7. A grid used in a colleague’s old C++ code (2005-2009)

8. A grid interface used to wrap both our in-house (2) and Dune (3) grids, still used in the steady-state upscaling codes (2009-today)

9. An interface designed to both deal with Dune (3) and UnstructuredGrid(4), similar to (5), but abandoned since (5) was better for the parallel case.
What have I learned?

• Grid interfaces are too much fun to write
• I’ll never find one that satisfies me 100%
  • I’d love to have one based on algebraic topology notation
• Do not generalize/abstract/wrap code until you are certain it is necessary
• Use the Dune interface, unless you really need something it does not provide
  • Random access, please...
Flow: FV order 1, upwind weighting

Requires:
- Connectivity graph
- Transmissibilities on graph edges (==faces)
- Cell depths and volumes

Ideal grid interface: only the above

High flexibility:
- Manipulate transmissibilities (faults)
- Manipulate connectivity graph (nnc, fake aquifers)
- Agnostic to actual grid type (CP, PEBI etc.)

Upscaling: mimetic method

Requires:
- Grid that is a cell-complex
- Face areas and centroids
- Cell volumes and centroids

Ideal grid interface: a cell-complex interface

Can support other discretizations:
- Higher-order methods
- Streamline methods
- Virtual Element methods (and some FE)
How can we eat our cake and have it too? (1)

Sketch of an idea:

<table>
<thead>
<tr>
<th>Finite Volume codes (discr. ops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible manipulations</td>
</tr>
<tr>
<td>Simple graph layer</td>
</tr>
<tr>
<td>Advanced discretizations</td>
</tr>
<tr>
<td>Cell-complex grid</td>
</tr>
<tr>
<td>• Parallel and adaptive</td>
</tr>
</tbody>
</table>

Some problems with this:

• Manipulations restrict adaptivity or vice versa
• Simple graph layer must also be parallel, partitioning must be done taking manipulations into account
How can we eat our cake and have it too? (2)

Another idea:

Finite Volume codes (discr. ops) & Advanced discretizations & Cell-complex grid & Parallel and adaptive & Additionally allows *fake neighbours*

Issues with this idea:
- Manipulations still restrict adaptivity or vice versa
Making Flow perform well
What is the main bottleneck?

A. Assembly of nonlinear equations?
B. Solving linear systems?
C. Input/output?
D. Other things?

Answer changes over time!

For OPM Flow and our target problems, always A or B.

(I/O performance has also been improved 3x)
## Bottleneck 1

**Linear solver horrendously slow**
- UMFPACK, direct solver
- Works for very small systems (SPE1)
- Breaks down for a few thousand cells

**Root cause:** linear system not well suited for direct solver

**Root cause:** direct solvers do not scale well

<table>
<thead>
<tr>
<th>Conserve W</th>
<th>Pressure</th>
<th>Water sat</th>
<th>Gas mix/s</th>
<th>Well flux</th>
<th>Well bhp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$W_p$</td>
<td>$W_{sw}$</td>
<td>$W_x$</td>
<td>$W_q$</td>
<td>$W_{bhp}$</td>
</tr>
<tr>
<td>Conserve O</td>
<td>$O_p$</td>
<td>$O_{sw}$</td>
<td>$O_x$</td>
<td>$O_q$</td>
<td>$O_{bhp}$</td>
</tr>
<tr>
<td>Conserve G</td>
<td>$G_p$</td>
<td>$G_{sw}$</td>
<td>$G_x$</td>
<td>$G_q$</td>
<td>$G_{bhp}$</td>
</tr>
<tr>
<td>Well flow</td>
<td>$Q_p$</td>
<td>$Q_{sw}$</td>
<td>$Q_x$</td>
<td>$Q_q$</td>
<td>$Q_{bhp}$</td>
</tr>
<tr>
<td>Well control</td>
<td></td>
<td></td>
<td></td>
<td>$C_q$</td>
<td>$C_{bhp}$</td>
</tr>
</tbody>
</table>
Bottleneck 1 – addressed

Use Schur complement to eliminate well unknowns

Use iterative solvers from Dune

Use 2-stage CPR preconditioner
  • Solve almost-elliptic system for pressure (with AMG precond.)
  • Solve full system with ILU0 precond.

Results:
  • SPE9 runtime 3 minutes (was 30 min)
  • Norne case ~6 times Eclipse runtime
Bottleneck 2

Assembly of nonlinear equations slow
  • Functions implement residual equations
  • AD class produces Jacobians

Root cause: simple operations too expensive
  • Every +/-* op triggers sparse matrix creation
  • Even when matrix is diagonal or identity!

\[
\text{flux[phase]} = \text{upwind.select}(b * \text{mob}) * (\text{transi} * \text{dh});
\]

Every multiplication, assignment and select() trigger sparse matrix creation.
Bottleneck 2 – addressed

Replace SparseMatrix in AD class with smart wrapper

- Wrapper treats zero, identity and diagonal matrices with custom code
- No change at all to the simulation code!

Result:
- Norne case ~3.5 times Eclipse runtime

```plaintext
flux[phase] = upwind.select(b * mob) * (transi * dh);
```

Now only select() trigger sparse matrix creation (since result depends on unknowns in multiple cells)
Bottleneck 3

Linear solver dominates runtime (again)

- Time-consuming setup of matrices for preconditioner and solver
- Outer linear solve of full system is slow
Bottleneck 3 – addressed

Change system matrix structure

- Use block-ILU0 instead of CPR
- Before: 3x3 system of NxN sparse matrices
- Now: NxN sparse matrix of 3x3 blocks (or 4x4 for polymer etc.)

Result:
- Norne case ~2.5 times Eclipse runtime
Bottleneck 4

Assembly of residual and Jacobians dominate (again)

Root cause: cache-unfriendly use of AD class

Root cause: (still) too many sparse matrix ops

\[ \text{flux[phase]} = \text{upwind.select(b \ast mob)} \ast (\text{transi} \ast \text{dh}); \]

The multiplication "b \ast mob" requires writing the result vector to memory before doing the next operation
Bottleneck 4 – addressed

Completely change assembly approach to use local AD
  • Meaning: only handle fixed number of local derivatives for each variable
  • Much better cache performance
  • Matrix assembly is separate
  • Clever trick to get derivatives for fluxes (that depend on two cells)
  • Was gradually prototyped by A. Lauser for 2-3 years before switching

Results:
  • Norne case ~1.7 times Eclipse runtime (~1.1 or better by now)

Consequences:
  • Assembly no longer resembles MRST
  • More complex code structure to understand for programmers
How to find the next bottleneck?

- Common sense and intuition? Often fails...
- Linuxbenchmarking.com
- Profiling:

<table>
<thead>
<tr>
<th>Weight</th>
<th>Self Weight</th>
<th>Symbol Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.97 min 100.0% 5.00 ms</td>
<td>▼Omp::SimulatorReport Omp::BlackoilModelEbos<a href="">Ewoms::Properties::TTag::EclFlowProblem</a>::nonlinearIteration&lt;Omp::NonlinearSolverEbos<a href="">Ewoms::Properties::TTag::EclFlowProblem</a>::assemble(Omp::SimulatorTimerInterface const&amp;, int) flow</td>
<td></td>
</tr>
<tr>
<td>7.43 min 49.6% 4.00 ms</td>
<td>▼Omp::BlackoilModelEbos<a href="">Ewoms::Properties::TTag::EclFlowProblem</a>::solveJacobianSystem(Dune::BlockVector&lt;Dune::FieldVector&lt;double, 3&gt;, 3&gt;, s) flow</td>
<td></td>
</tr>
<tr>
<td>7.32 min 48.8% 1.00 ms</td>
<td>▼Omp::BlackoilModelEbos<a href="">Ewoms::Properties::TTag::EclFlowProblem</a>::getConvergence(Omp::SimulatorTimerInterface const&amp;, int, std::vector&lt;int&gt; const&amp;) flow</td>
<td></td>
</tr>
<tr>
<td>9.43 s 1.0% 0 s</td>
<td>▼Ewoms::BlackOilNewtonMethod<a href="">Ewoms::Properties::TTag::EclFlowProblem</a>::update_(Dune::BlockVector&lt;Ewoms::BlackOilPrimaryVariables<a href="">Ewoms::Properties::TTag::EclFlowProblem</a>::_platform_bzero$VARIANT$Haswell libsyste platform dylib</td>
<td></td>
</tr>
<tr>
<td>3.20 s 0.3% 0 s</td>
<td>▼Omp::NonlinearSolverEbos&lt;Ewoms::Properties::TTag::EclFlowProblem, Omp::BlackoilModelEbos<a href="">Ewoms::Properties::TTag::EclFlowProblem</a>::_platform_bzero$VARIANT$Haswell libsyste platform dylib</td>
<td></td>
</tr>
<tr>
<td>747.00 ms 0.0% 3.00 ms</td>
<td>▼std::chrono::steady_clock::now() libc++.1.dylib</td>
<td></td>
</tr>
<tr>
<td>208.00 ms 0.0% 208.00 ms</td>
<td>▼free large libsystem malloc.dylib</td>
<td></td>
</tr>
<tr>
<td>65.00 ms 0.0% 3.00 ms</td>
<td>▼operator new(unsigned long) libc++abi.dylib</td>
<td></td>
</tr>
<tr>
<td>8.00 ms 0.0% 2.00 ms</td>
<td>▼free libsystem malloc.dylib</td>
<td></td>
</tr>
<tr>
<td>5.00 ms 0.0% 0 s</td>
<td>▼Omp::WellCollection::groupControlActive() const flow</td>
<td></td>
</tr>
<tr>
<td>4.00 ms 0.0% 0 s</td>
<td>▼Omp::Logger::addMessage(long long, std::basic_string&lt;char, std::char_traits&lt;char&gt;, std::allocator&lt;char&gt; &gt; const&amp;, const) const flow</td>
<td></td>
</tr>
<tr>
<td>3.00 ms 0.0% 1.00 ms</td>
<td>▼DYLD-Stub$S_MPI_Allreduce flow</td>
<td></td>
</tr>
<tr>
<td>2.00 ms 0.0% 2.00 ms</td>
<td>▼Omp::SimulatorReport::operator+=(Omp::SimulatorReport const&amp;) flow</td>
<td></td>
</tr>
<tr>
<td>2.00 ms 0.0% 0 s</td>
<td>▼Omp::UgGridHelpers::numCells(Dune::CpGrid const&amp;) flow</td>
<td></td>
</tr>
<tr>
<td>1.00 ms 0.0% 1.00 ms</td>
<td>▼free_tiny libsystem malloc.dylib</td>
<td></td>
</tr>
</tbody>
</table>
Technology for a better society