OP2-Clang: A source-to-source translator using Clang/LLVM LibTooling

Gábor Dániel Balogh, Dr. Gihan Mudalige, Dr. István Reguly

Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

June 21, 2018

Outline

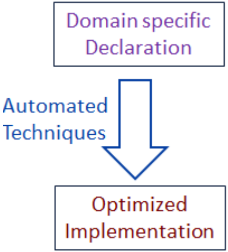
- Motivation
- Unstructured grids
- OP2
  - Abstraction
  - API
- Source-to-source transformation with clang
- OP2-Clang and Skeletons
- Performance results

Future proofing parallel HPC applications

- Hardware is rapidly changing with ambitions to overcome exascale challenges
- There is considerable uncertainty about which platform to target
  - Not clear which architectural approach is likely to “win” in the long-term
  - Not even clear in the short-term which platform is best for each application
- Increasingly complex programming skills set needed to extract best performance for your workload on the newest architectures.
  - Need a lot of platform specific knowledge
  - Cannot be re-coding applications for each “new” type of architecture or parallel system.

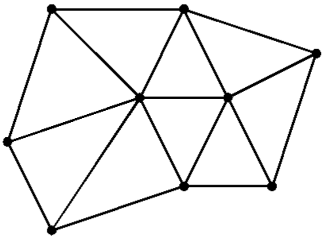
One approach to develop future proof HPC applications is the use of domain specific high-level abstractions (HLAs)

- Provide the application developer with a domain specific abstraction
  - To declare the problem to be computed
  - Without specifying its implementation
  - Use domain specific constructs in the declaration
- Create a lower implementation level
  - To apply automated techniques for translating the specification to different implementations
  - Target different hardware and software platforms
  - Exploit domain knowledge for better optimisations on each hardware system



Unstructured grids

- A collection of nodes, edges, etc., with explicit connections - e.g. mapping tables define connections from edges to nodes
- Harder to parallelize due to connections and dependencies
- Hard to avoid race conditions



- PDEs can be easily mapped to algorithms on unstructured meshes
- For many interesting cases, unstructured meshes are the only tool capable of delivering correct results

OP2

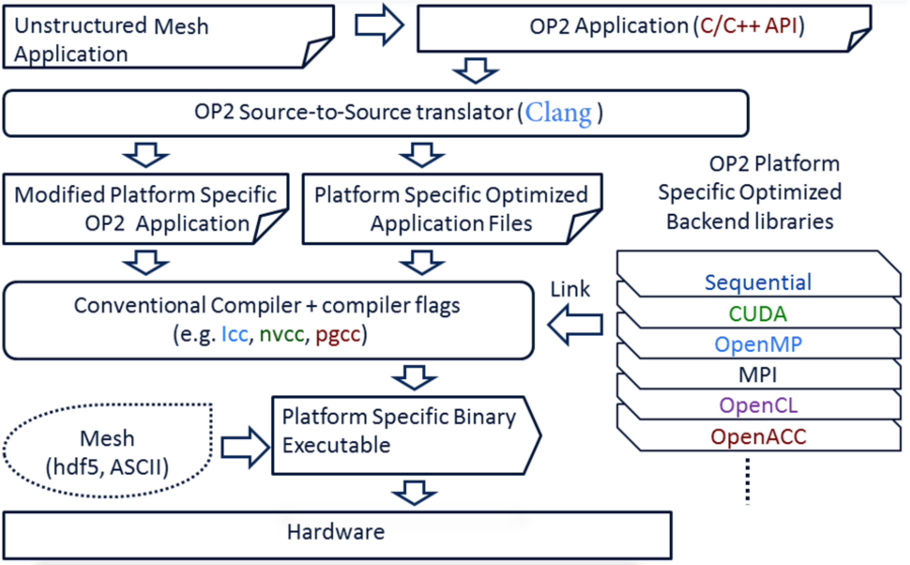
- Open Source project
- OP2 based on OPlus (**O**xford **P**arallel **L**ibrary for **U**nstructured **S**olvers), developed for CFD codes on distributed memory clusters
- Support application codes written in C++ or FORTRAN
- Looks like a conventional library, but uses code transformations (source to source translator) to generate parallel codes

OP2 Abstraction

- Sets (e.g. nodes, edges, faces)
- Datasets on sets (e.g. flow variables)
- Mappings (e.g. from edges to nodes)

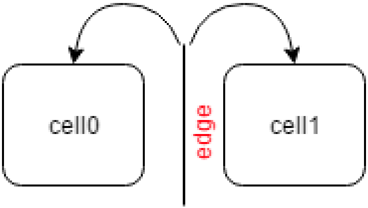
- Parallel loops
  - Operate over all members of one set
  - Datasets accessed at most one level of indirection
  - User specifies how data is used (e.g. Read-only, write-only, increment, read/write)

- Restrictions
  - Set elements can be processed in any order, doesn't affect results within machine precision
  - Static sets and mappings (no dynamic grid adaptation)


Unstructured Mesh Application
OP2 Application (C/C++ API)
OP2 Source-to-Source translator (Clang)
Modified Platform Specific OP2 Application
Platform Specific Optimized Application Files
OP2 Platform Specific Optimized Backend libraries
Conventional Compiler + compiler flags (e.g. Icc, nvcc, pgcc)
Link
Sequential
CUDA
OpenMP
MPI
OpenCL
OpenACC
Mesh (hdf5, ASCII)
Platform Specific Binary Executable
Hardware

OP2 loop over edges

```
void res(double* edge,
         double* cell0,
         double* cell1){
   *cell0 += *edge;
   *cell1 += *edge;
}
```




```
op_par_loop(res,"residual_calculation", edges,
  op_arg(dedges, -1, OP_ID, 1, "double", OP_READ,
  op_arg(dcells, 0, pecell, 1, "double", OP_INC),
  op_arg(dcells, 1, pecell, 1, "double", OP_INC));
```

Clang LibTooling for code generation

- Gives direct support for source-to-source transformations (Tooling/Refactoring)
- Nice and robust abstraction for local changes in the source code
    - Search in the AST for interesting bits of code with the ASTMatchers interface
    - Based on the location of the match create patches to the source code
- Hard to handle significant structural transformations

The code transformation divided to two steps:

- Collecting data and modifying the user given OP2 application files
- Generating target specific implementations for the computational loops
    - Target specific implementations are significantly different from the user functions

The generated code for different loops are very similar in OP2

- A lot of static code in the generated loop
- We need local changes only to transform a skeleton application to perform the given operation

```
void skeleton(double d) {}

void op_par_loop_skeleton(char const *name, op_set set,
                          op_arg arg0) {

  int nargs = 1; op_arg args[1] = {arg0};
  int exec_size = op_mpi_halo_exchanges(set, nargs, args);

  for ( int n = 0; n < exec_size; n++ ){
    if (n == set->core_size) op_mpi_wait_all(nargs, args);

    int map0idx = arg0.map_data[n * arg0.map->dim + 0];


    skeleton(&((double *)arg0.data)[2 * map0idx]);
  }
}
```

Kernel function

Number of arguments

Static code

Prepare indirect accesses

Set up pointers, call kernel

Generated code for the example loop

```
void res(double* edge,double* cell0,double* cell1) {
    *cell0 += *edge; *cell1 += *edge; }

void op_par_loop_res(char const *name, op_set set,
                     op_arg arg0, op_arg arg1,
                     op_arg arg2) {

  int nargs = 3; op_arg args[3] = {arg0, arg1, arg2};
  int exec_size = op_mpi_halo_exchanges(set, nargs, args);

  for ( int n = 0; n < exec_size; n++ ){
    if (n == set->core_size) op_mpi_wait_all(nargs, args);

    int map0idx = arg0.map_data[n * arg0.map->dim + 0];
    int map1idx = arg0.map_data[n * arg0.map->dim + 1];


   res(&((double *)arg0.data),
       &((double *)arg1.data)[2 * map0idx],
       &((double *)arg1.data)[2 * map1idx]);
  }
  // …
}
```

Kernel function

Number of arguments

Static code

Prepare indirect accesses

Set up pointers, call kernel

Static code

The base of the transformation - ASTMatchers

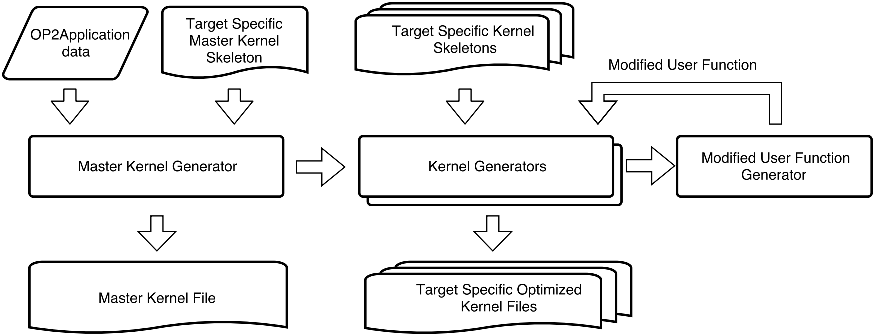
```
void op_par_loop_skeleton(...) {
  // …
  for (int n = 0; n < exec_size; n++){
    // …
    skeleton(
      &((double*)arg0.data)[2*map0idx]);
  }
  // …
}
```

```
callExpr(callee(functionDecl(
  hasname("skeleton"))))
  .bind("function_call");
```

```
Generating replacement for
key "function_call"
```

```
void op_par_loop_skeleton(...) {
  // …
  for (int n = 0; n < exec_size; n++){
    // …
    res(&((double *)arg0.data),
      &((double *)arg1.data)[2 * map0idx],
      &((double *)arg1.data)[2 * map1idx]);
  }
  // …
}
```

Kernel generation process using skeletons



Advantages of the skeleton approach

- Easy to extend with new target
  - Writing the skeleton is similar to write a simple loop
  - Matchers and callbacks can be reused
- More robust code generation
  - We search in the AST the static part is checked
  - The only source of errors are the generated parts

Airfoil and Volna

- Airfoil
  - Non-linear 2D inviscid airfoil code
  - Five kernels with different access patterns:
    - **save_soln** - simple kernel, only direct reads and writes
    - **adt_calc** - computationally expensive operations, indirect reads, direct increments
    - **res_calc** - complex computation, indirect reads and indirect increments
    - **bres_calc** - similar to **res_calc** but on the boundary edges
    - **update** - simple computation with a global reduction, only direct reads and writes
- Volna
  - Shallow water simulation capable of handling the complete life-cycle of a tsunami
  - Most time consuming kernels:
    - **SpaceDiscretization** - indirect reads and increments
    - **NumericalFluxes** - indirect reads and global reduction
    - **computeFluxes** - indirect reads

Airfoil and Volna performance

	Speedup with Vectorization vs Sequential	Speedup with OpenMP (with 16 cores) vs Sequential	Speedup with CUDA (P100) vs OpenMP
Airfoil	2.08	10.33	4.28
Volna	2.34	12.9	3.46

Summary

- OP2 abstraction facilitate the development of application for parallel execution
- Nearly optimal performance
  - but the optimization is done automatically, not by the developer
- OP2-Clang generates multiple parallelized implementations for applications
  - OpenMP, Vectorized, CUDA
- With the introduction of parallelization skeletons the transformations became simple local transformations.
  - The code generation much simpler and robust
  - Easy to add new parallelizations, optimizations with adding new skeletons