Icosahedral Modeling with GGDML

Nabeeh Jum'ah[1], Julian Kunkel[2], Günther Zängl[3], Hisashi Yashiro[4], Thomas Dubos[5], and Yann Meurdesoif[6]

1 Universität Hamburg Jumah@informatik.uni-hamburg.de, 2 Deutsches Klimarechenzentrum, 3 Deutscher Wetterdienst, 4 RIKEN Advanced Institute for Computational Science, 5 École Polytechnique, 6 LSCE

Abstract

The atmospheric and climate sciences and the natural sciences in general are increasingly demanding for higher performance computing. Unfortunately, the gap between the diversity of the hardware architectures that the manufacturers provide to fulfill the needs for performance and the scientific modeling can not be filled by the general-purpose languages and their compilers. The scientists who develop the models need to manually optimize their models to exploit the capabilities of the underlying hardware that will run the model. This needs providing multiple versions of the code, or at least some parts of it, when running the model on a different machine. This is not a trivial problem when heterogeneous computing infrastructures are being considered to support the exascale computing era.

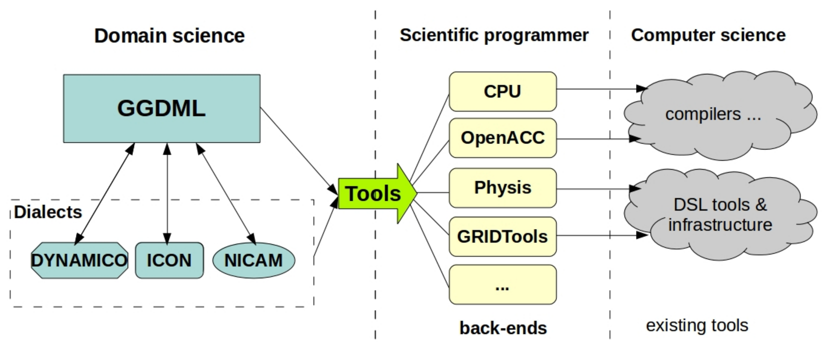
In order to provide performance porability to the icosahedral climate modeling we have developed a set of higher-level language extensions that we call GGDML. The extensions provide semantically-higher-level constructs with which scientists can express their scientific problem with scientific concepts. This eliminates the need to explicitly provide lower-level machine-dependent code. With this solution, scientists still use the general-purpose language. The parts of the code in which a scientist uses the GGDML extensions are translated by asource-to-source translation tool that optimizes the generated code to a specific machine. The translation process is driven by configurations that are provided indpendantly from the source code.

In this poster we review some GGDML extensions and we focus mainly on the configurable code translation of the higher-level code.

Goals

With the approach that we suggest we aim at an enhanced and more productive software development process through which a single source code that is easily maintainable can be developed. The source code is mainly developed with the general-purpose language that the developer scientists choose for modeling. The code that would eventually run on a machine would exploit its performance-supporting capabilities. The software development process fosters speration of concerns:

- Scientists from the domain science provide the problem logic in terms of scientific concepts.
- The configuration that is responsible for platform-dependent implementation is provided by scientific programmers.



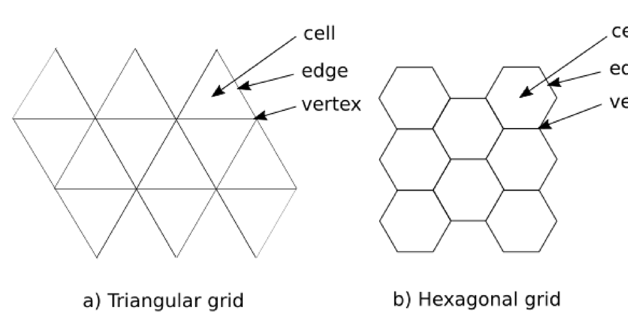
GGDML Extensions

GGDML *(General Grid Definition and Manipulation Language)* provides abstract grid concepts that support unstructured grids like icosahedral models besides to regular grids.

GGDML has been developed in a co-design approach in collaboration with domain scientists.

The set of GGDML extensions

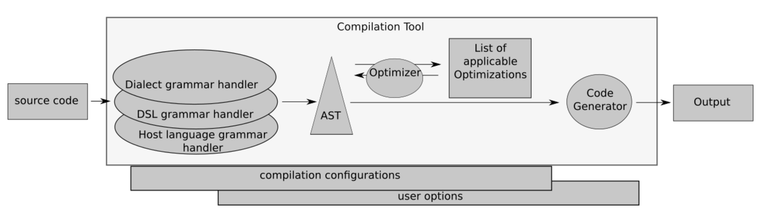
- Extends a general-purpose language
  - It extends the grammer of the language
  - The concept applies to the different languages in general
- Allows for the definition of grids
  - Various shapes, e.g., triangular, hexagonal



- Allows to define variables on the grid
- Allows to reference variables by grid elements
  - Named element relationships
    - to reference cell edge
    - to reference cell above/below
    - to reference a neighbour cell
    - ...
- Provides an iterator to traverse the grid
  - Specify/modify dimensions of ranges
  - Update data of variables while traversing
- Provides a reduction operator

Code Translation

The higher-level code is translated into the modeling language. A lightweight source-to-source translation tool that ships with the code repository does that.



Translation Configuration

- Allows to control the way of the variable declaration on the grid.
  - This is handled by specifying the extensions within the configuration file
  - Groups of alternatives that are provided by the configuration control the variable declaration
    - 2D/3D group to control the dimensionality of the grid
    - CELL/EDGE/VERTEX to control the grid parts where the variable is measured/computed
  - The groups and the alternatives are dynamic, they can be changed/expanded on need

```
SPECIFIERS: SPECIFIER(loc=CELL|EDGE) SPECIFIER(dim=3D|2D)
```

- Allows to control the way the variables are allocated/deallocated
  - Full control over the code to do the memory allocation/deallocation
  - The configuration gives the flexibility to use different allocation/deallocation codes for variables based on the different declaration options.
    - The allocation/deallocation configuration sections provides declaration groups and alternatives as parameters to write different codes for different variable contexts
    - e.g. a variable that is declared with the $3D$ option for the *dimension* group uses an allocation code different from that of $2D$

```
ALLOCATIONS:
 CASE loc=CELL & dim=3D:
 $var_name = ($data_type*restrict)malloc(
             g->cBlkCnt*g->hight*g->blkSize*sizeof($data_type));
```

- Provides the way to specify the default dimensions of the grid and its components
  - Can serve to define structured and unstructured grids
  - The different components of the grid are configurable, for example
    - The set of the cells of the 3D grid (same for edges,vertices, or whatever component needed)
    - The set of the cells on the surface (2D)
  - The default grid specifications can be overrided in an iterator for a specific kernel

```
GLOBALDOMAIN:
 COMPONENT(CELL3D):
  RANGE OF hight= 0 TO g->hight
```

- Allows to define iterator index operators to enable an improved grid traversal
  - The operators are dynamically defined in the configuration files
  - This provides high flexibility to make the extensions fit different domain and application-specific needs, for example
    - In a hexagonal icosahedral grid we can define the operator *upright* to access the neighboring cell at the upper right direction
    - In a triangular icosahedral grid we can define *neighbor* with a numer parameter to refer to one of the three neighboring cells
    - In a regular grid we can define *right* or *east* to refer to the cell on the right
  - The operators then modify the iterator's index by some calculation or using a precomputed array (like in icosahedral grids connectivity)

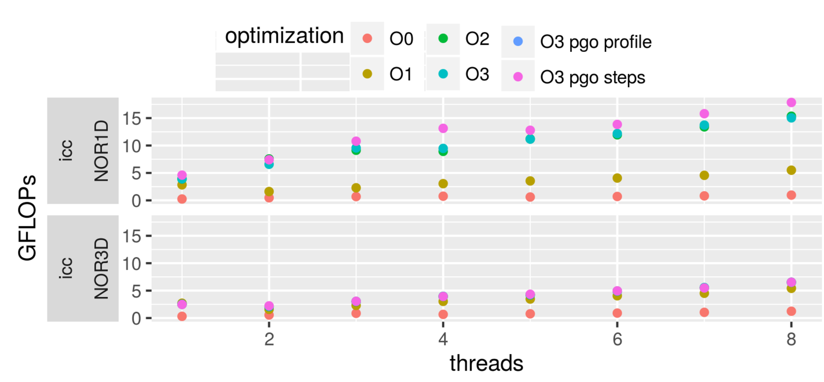
```
INDEXOPERATORS:
 above(): hight=$hight+1
```

- Allows to define the memory layout of the variables declared with the extensions
  - The configuration allows different layouts by changing
    - the memory layout when allocating the variables
    - generating the right indices when accessing a variable in an iterator
  - The configuration allows index transformation with general-purpose language expressions or even transformation functions (e.g. Hilbert filling curve)
  - The configuration also provides the ways to exchange data with fixed memory layout arrays
- Allows to control code annotation
  - This allows the configuration to guide annotating code with OpenMP or OpenACC for example

Memory Layout and Performance Impact

The figure (right) shows the impact of changing the memory layout of an application with various optimization options with intel compiler(on an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz machine).

The table below shows the impact of changing the memory layout of a stencil code of 3,5, and 7-point stencils on CPUs(Ivy Bridge E5-2690 v2 3.0GHz) and GPUs(Nvidia K80 and P100) –PGI compiler.



Stencil	CPU Performance (GFlops/s)		K80 GPU Performance (GFlops/s)		P100 GPU Performance (GFlops/s)	
	Normal 3D array	1D addressing	Normal 3D array	1D addressing	Normal 3D array	1D addressing
5	71	72	78	128	189	342
7	97	97	93	169	243	394
9	112	117	102	195	287	431

Code Example

The following example demonstrates the use of GGDML for vertical integration.

```
FOREACH cell IN grid
{
    gv_vi[cell] += gv_temp[cell];
}
```

Translating the code for a CPU with OpenMP annotation the following code is generated:

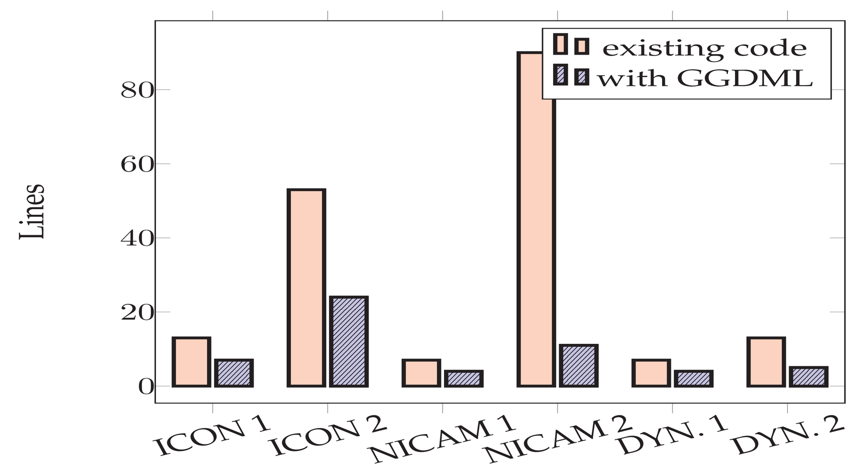
```
 #pragma omp parallel for
for(int block_index = ( 0); block_index < (
     g->cBlkCnt) ; block_index++) {
for(int hight_index = ( 0); hight_index < (
     g->hight) ; hight_index++) {
for(int cell_index = ( 0); cell_index < (
     g->blkSize) ; cell_index++) {
 gv_vi[ ( block_index)][ ( cell_index)] +=
      gv_temp[ ( block_index)][ ( hight_index)][ (
      cell_index)] ;
}
}
}
```

Translating the same code for a GPU with OpenACC annotation and 1D-transformed memory layout the following code is generated:

```
 #pragma acc parallel loop gangs
for(int block_index = ( 0); block_index < (
     g->cBlkCnt) ; block_index++) {
 #pragma acc loop worker
for(int hight_index = ( 0); hight_index < (
     g->hight) ; hight_index++) {
 #pragma acc loop vector
for(int cell_index = ( 0); cell_index < (
     g->blkSize) ; cell_index++) {
 gv_vi[ ( block_index) * g->blkSize + (
      cell_index)] += gv_temp[ ( block_index) *
      g->blkSize * g->hight + ( hight_index) *
      g->blkSize + ( cell_index)] ;
}
}
}
```

Code Quality

We have previously taken two relevant kernels from each of the three icosahedral models: ICON,Nicam, and Dynamico, and analyzed the achieved code reduction. The figure below gives an indication for that.



- In average, we cut down the LOC to (30%) of the original code. Better reductions are achieved in stencil codes (NICAM example No.2, reduced to 12.22% of the original LOC).
- Code reduction reduces development time and costs. By applying COCOMO to a case model we estimated a cost reduction from 12.3 to 5.7 M€ for a project with semi-detached team and from 6.5 to 3.1 M€ for organic team.

Summary

- GGDML extensions provide a way to improve climate/atmospheric models development
- GGDML extensions lift the model development process to a higher level that enables improved code maintainability & readability while providing performance portability.
- GGDML and the translation technique eliminate the need for lower-level architecture-specific details in the source code.
- GGDML significantly reduces the size of the source code and model development costs.
- A target-specific configuration (independant of the source code) guides the generation of a machine-dependent optimized code.
- Scientist do not need to care about computing details, scientific programmers write the configuration that leads the optimization process.
- The whole process is controlled and driven by the users, thanks to the configuration flexibility which allows to define/redefine the language extensions.

Acknowledgements

This work was supported in part by the German Research Foundation (DFG) through the Priority Programme 1648 "Software for Exascale Computing" (SPPEXA) (GZ: LU 1353/11-1).