Referee Comment to GMD manuscript gmd-2014-107:
The generic simulation cell method for developing extensible, efficient and readable parallel computational models
by I. Honkonen

General impression

The paper discusses application of a programming technique realisable in the C++11 language for implementing data structures of use in computational modelling. These structures are intended for grouping properties of single grid cells that are part of a computational domain. The examples presented in the paper include geoscience-relevant ones such as prototype advection equation solver and its coupling with particle-tracking logic.

Technically speaking, the discussed approach is an application of a C++11 variadic-template-based idiom for defining data structures which store not only values but also meta-data that are relevant for handling communications in a distributed-memory framework. The crux of the presented approach lies in offering maintainable and extendable way of coding compile-time logic for interrogation on these meta data. The maintainability and extendabilty aspects are noticeable when coupling multiple software packages (assuming all are developed using the presented technique).

Discussion on the applicability of the idiom is based on several example programs briefly described in the paper. The complete code of the first of the example programs is presented in the paper with two listings. Subsequent example programs have accompanying listings that cover only several snippets from the code.

All example programs are based on an open-source header-only C++ library developed by the author. The library depends on Boost C++ library and the C version of the Message Passing Interface (MPI). All codes but the first example are dependant also on an external library "dccrg" introduced in a previously-published paper by the author and co-workers (Honkonen et al. 2013, Comp. Phys. Comm. 184).

The topic seems quite uncommon for GMD judging by the contents of other papers in the journal, but my personal opinion is that such material matches the title (and hence the scope) of the journal – geoscientific model development. Let me underline that I personally like the idea of the paper, that is to focus the discussion on a particular programming technique relevant to geoscientific modelling.

It is worth underlining that the author develops and disseminates the code in a way that facilitates its use, comprehension, review and further extensions:

- by providing public access to the code with no technical or legal obstacles,
- by structuring the code as a relatively standalone package,
- by adhering to numerous coding conventions that enhance code readability.

In the following section I list three main points that I see worth addressing before publishing the paper in GMD.

Main remarks

1. In my opinion, the paper lacks a simple example that would clearly exemplify the advantages of the presented approach. My suggestion hence is to begin the paper with presentation of two complete simple examples implemented (i) using the idiom/library and (ii) without using it. Comparing such two compact codes should help to identify the advantages of the version that uses the library. Currently, the first example only shows that using the library does not make a program more complex to read, but it does not show any benefit of using it. If employing MPI is essential to show it, the introductory example should employ MPI. Furthermore, to make it feasible for a reader to comprehend this introductory example, it should reveal all calls to the library interface – currently in the parallel examples these calls are hidden from the reader as they are located within the external "dccrg" library. Let me underline, that this concerns only a first ("hello-world") example – all subsequent more complex examples shall employ other dependencies, and "dccrg" in particular. Generally, I suggest not to

overestimate the readers' abilities to grasp the ideas of the more advanced examples just from partial listings and textual description, especially as they are highly dependant on the "dccrg" logic discussed elsewhere.

2. A second issue I find important to address at the very beginning of the paper, is to define clearly the elemental assumptions of the library design and its implications. In my understanding, these are in particular the cell-based memory layout, parallelism using domain-decomposition and the dependency on MPI (and its communication model). Stating these assumptions clearly at the beginning (and perhaps discussing them further in subsequent sections) is essential to understand the idea, the limitations and possible applications of the library.

   The cell-based memory layout implies that values of a single property of all cells are not laid out contiguously in memory (they are interleaved by other properties of the cells). Taking into account that the presented examples employ rectangular grids and stencil algorithms (game of life, advection), it seems worth discussing how such design choice may influence the algorithm performance and/or code readability (apparently this discussion could be based on the already implemented game_of_life/non_cellular.cpp example).

   The MPI communication model harmonises well with multi-node distributed-memory setups. It is worth mentioning how the current design influences the possibilities to leverage the shared-memory parallelism available with multi-core CPUs.

3. The third point is slightly subjective. In the introduction, the author pleasantly immerses the reader into the realm of modern object-oriented coding aimed at creating reliable and maintainable software. Yet, one may feel surprised that subsequent sections evade discussion on how the introduced library design limits one in applying some constructs that would intuitively be expected to be attainable with an object-oriented library. Notable examples are loop-free constructs and indirection mechanisms typical for object-oriented data containers. Such features can certainly be left to be added to the library later (also by the users, as it is free software). Yet, discussing it would help the reader to build up an understanding of what can and cannot be achieved with the library.

In the subsequent section I present two code listings written while getting acquainted with the library. I include them here just to illustrate the statements from the above points and from some of the technical comments given at the end of the report.

Alternative serial Game-of-Life implementations illustrating the above points

The two listings below depict two example alternative implementations of the Game of Life program presented in the paper in Figure 2. The listing **A** on the left follows closely the listing from Figure 2 in the paper but does not use gensimcell. The listing **B** on the right also does not use gensimcell but employs the Blitz++ library instead of std::array<std::array<>> what allows on to use several readability-improving constructs.

- Listing **A** achieves the same what the one in Figure 2 does, yet without using gensimcell; I do understand that it was the aim of the author to show that using gensimcell in a serial simulation does not change the code much, but it leaves the reader puzzled as to why the variadic-template-based structure is any better than an ordinary one; I strongly suggest to supplement the paper with a complete example that will show an advantage of employing the presented approach.
- Listing **B** shows how employment of an object-oriented data-container library Blitz++ allows one to make the code from listing **A** a bit shorter and arguably more readable[1]:
  - there is no need to define a "print_game()" function to output the model state (as done in examples/game_of_life/serial.cpp);

[1] note that Blitz++ does not allow for structures of heterogeneous datatypes, hence Is_Alive is defined as integer and not a bool

– some stages of the algorithm can be coded without loops over grid dimensions (cf. the "set new state" stage);

– one may work on an array of one of the components even when the cell-based storage is used (e.g. grid[Is_Alive](x,y), cout << grid[Is_Alive]).

All these would arguably be expected to be attainable when implementing such algorithm in an object-oriented way (analogous mechanisms work for instance in NumPy). Yet, none of such features is used in the paper – apparently either due to the gensimcell design or limitations of the "std::array<std::array<>>" manually-multidimensionalised container. Discussion of it could help the reader in understanding the applicability range of the presented approach.

A: GoL example without gensimcell

```
#include <array>
#include <cstdlib>

struct Cell_T { bool Is_Alive; int Live_Neighbours; };

int main() {
  const int width = 10, height = 10;
  std::array<std::array<Cell_T, width>, height> grid;

  // initial condition
  for (auto& row : grid) {
  for (auto& cell : row) {
    cell.Live_Neighbours = 0;
    cell.Is_Alive = rand() < RAND_MAX / 10;
  }}

  for (int step = 0; step < 20; ++step) {
    // collect live neighbour counts
    for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
      auto& cell = grid[row][col];

      // neighbour index offsets: +1, 0, -1 (+size)
      for (auto row_offset : {1, 0, width - 1}) {
      for (auto col_offset : {1, 0, height - 1}) {
        if (row_offset == 0 && col_offset == 0)
          continue;

        const auto& neighbor = grid[
          (row + row_offset) % height
        ][
          (col + col_offset) % width
        ];

        if (neighbor.Is_Alive)
          cell.Live_Neighbours++;
      }}
    }}

    // set new state
    for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
      Cell_T& cell = grid[row][col];

      if (cell.Live_Neighbours == 3)
        cell.Is_Alive = true;
      else if (cell.Live_Neighbours != 2)
        cell.Is_Alive = false;

      cell.Live_Neighbours = 0;
    }}
  }
}
```

B: GoL example w/o gensimcell using Blitz++

```
#include <blitz/array.h>
#include <random>

const int Is_Alive = 0, Live_Neighbours = 1;
struct Cell_T { int Is_Alive, Live_Neighbours; };
BZ_DECLARE_MULTICOMPONENT_TYPE(Cell_T, int, 2);

int main() {
  const int width = 10, height = 10;
  blitz::Array<Cell_T, 2> grid(height, width);

  // initial condition
  {
    std::random_device rd;
    std::default_random_engine gen(rd());
    std::uniform_real_distribution<> dis(0,10);
    for (auto& cell : grid) cell.Is_Alive = dis(gen) < 1;
  }
  std::cout << grid[Is_Alive];

  for (int step = 0; step < 20; ++step) {
    // collect live neighbour counts
    for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {

      // periodic boundaries
      struct mod { int operator()(int i, int m) { return (i + m) % m; } };

      grid[Live_Neighbours](row, col) =
        grid[Is_Alive](mod()(row + 1, height), mod()(col + 0, width)) +
        grid[Is_Alive](mod()(row - 1, height), mod()(col + 0, width)) +
        grid[Is_Alive](mod()(row + 0, height), mod()(col + 1, width)) +
        grid[Is_Alive](mod()(row + 0, height), mod()(col - 1, width));
    }}

    // set new state
    // (Blitz++'s OOP expression-template loop-free notation)
    grid[Is_Alive] = where(
      grid[Live_Neighbours] == 3,  // if
      true,                        // then
      where(                       // else
        grid[Live_Neighbours] != 2,  // if
        false,                       // then
        grid[Is_Alive]               // else
      )
    );

    std::cout << grid[Is_Alive];
  }
}
```

Other comments (in random order))

- I suggest changing "The generic ..." into "A generic ..." in the title (as it is used in the paper text).
- Many GMD papers mention the program name in the title (i.e. "gensimcell") – it is not required with this manuscript type, but perhaps might be a good idea?
- I suggest mentioning domain decomposition and the geoscientific context in the abstract;
- Offering a snapshot of the code repository as an electronic supplement to the paper would make long-term archival of the paper viable – the code is essential to fully understand the text.

- I suggest summarising the library API in one place – that is listing all methods with their short description, as it is done in gensimcell_impl.hpp for Doxygen.
- Why not use "Listing" instead of "Figure"?
- I wonder if adding a figure with the simulation results from the first example would not help the presentation?
- Subsections 1.1 and 4.1 are the only subsections within their parent sections, both are preceded by an unnumbered body of text, restructuring into multiple subsection or creating new sections would be more logical.
- I would argue that a mention of Conway's Game of Life in a geoscientific journal requires a reference, Wikipedia and other sources suggest: Gardner 1970, Sci. Amer. 223(4)[2]
- I suggest listing all library dependencies at some place (Boost components, the C MPI API).
- The statement in line 18 on page 7/4583 that using gensimcell "adds hardly any code compared to a traditional implementation" is perhaps a good place to discuss that using it implies loop-based syntax.
- All mentions of "parallel computational model" seem to assume domain decomposition – this is misleading as parallelism may take also other forms.
- Perhaps it would be worth to mention that the [] operator is chosen arbitrarily in the implementation of the Cell class (Fig. 1).
- The listing in Figure 7 is syntactically incorrect – there are two structures with the same name defined.
- Introducing the name of the third state of a triboolean would prevent understanding "indeterminate" as a part of the sentence (BTW, there is a typo in source/gensimcell_impl.hpp: underemined).
- For simplicity (as with the const correctness), the private/public distinction may be omitted from the listings by defining all classes as structs (as done e.g. in Numerical Recipes).
- Describing the advection example, it is worth mentioning which algorithm it implements.
- In line with the third main comment and listing **B**: the presented examples feature numerous constructs or choices that seem not in line with the embraced object-orientation and modern C++11 standard:
  - the *std::array<std::array<>>* manually multi-dimensionalised containers are used while numerous C++ libraries offer multidimensional array containers with the significant advantages of: relieving the user from writing loops (even std::valarray), pre-defined input/otput iostream-compatible operators (e.g. Blitz++), boosting performance with temporary-array-free lazy evaluation of compound expressions (Blitz++, Eigen), caring about vectorisation and cache-friendliness (Eigen);
  - the C MPI interface is used while Boost.MPI offers an object-oriented one (as used in the dccrg library);
  - the C pseudorandom number generator is used while C++11 offers an object-oriented equivalent with <random> (see listing **B**).

  If there were reasons for not choosing any of those, these are certainly worth mentioning.
- The sentence "no changes to existing code are required for combining models" concerns models that where written in C++ using the very same library aimed at coupling models. It is misleading.
- The reference to Stroustrup 1999 in line 23 on page 4579 seems not a best match for the sentence as the cited paper does not even mention the template mechanism by name, perhaps

[2] `http://www.scientificamerican.com/article/mathematical-games-1970-10/`

citing Veldhuizen and Gannon 1998 ("Active libraries: Rethinking the roles of compilers and libraries") would be more appropriate?

- Why #include "" and not #include <> is used for system headers?
- Having in mind the level of generality of the discussion in the paper, I would suggest skipping the comments on copy-pasting code.
- Being so specific about the CPU type in section 5 calls for using the "-march=native" flag for compilation.
- The vague statement in section 5 that gensimcell "... should not slow down a computational model too much" could be exchanged with a one on scalability requirement;
- I suggest skipping "return 0" and "return EXIT_SUCCESS" – one line less, and this is the default behaviour.
- I suggest not naming something a ,,traditional scientific code" – it can mean a very different thing depending on the reader.

Technical corrections

- typo: varible (page 6/4582, line 20);
- typo: requred (page 15/4591, line 1);
- markup leftover: {API} in references;
- underscores are not readable in several places in the listings (e.g. MPI_...);

Hope that helps,
Sylwester Arabas