Geosci. Model Dev., 8, 473–483, 2015
www.geosci-model-dev.net/8/473/2015/
doi:10.5194/gmd-8-473-2015

Geoscientific Model Development
Open Access

A generic simulation cell method for developing extensible, efficient and readable parallel computational models

I. Honkonen[1,*]

[1]Heliophysics Science Division, Goddard Space Flight Center, NASA, Greenbelt, Maryland, USA
[*]previously at: Earth Observation, Finnish Meteorological Institute, Helsinki, Finland

Correspondence to: I. Honkonen (ilja.j.honkonen@nasa.gov)

Received: 11 June 2014 – Published in Geosci. Model Dev. Discuss.: 18 July 2014
Revised: 11 February 2015 – Accepted: 16 February 2015 – Published: 6 March 2015

Abstract. I present a method for developing extensible and modular computational models without sacrificing serial or parallel performance or source code readability. By using a generic simulation cell method I show that it is possible to combine several distinct computational models to run in the same computational grid without requiring modification of existing code. This is an advantage for the development and testing of, e.g., geoscientific software as each submodel can be developed and tested independently and subsequently used without modification in a more complex coupled program. An implementation of the generic simulation cell method presented here, generic simulation cell class (gensimcell), also includes support for parallel programming by allowing model developers to select which simulation variables of, e.g., a domain-decomposed model to transfer between processes via a Message Passing Interface (MPI) library. This allows the communication strategy of a program to be formalized by explicitly stating which variables must be transferred between processes for the correct functionality of each submodel and the entire program. The generic simulation cell class requires a C++ compiler that supports a version of the language standardized in 2011 (C++11). The code is available at https://github.com/nasailja/gensimcell for everyone to use, study, modify and redistribute; those who do are kindly requested to acknowledge and cite this work.

1 Introduction

Computational modeling has become one of the cornerstones of many scientific disciplines, helping to understand observations and to form and test new hypotheses. Here a computational model is defined as numerically solving a set of mathematical equations with one or more variables using a discrete representation of time and the modeled volume. Today the bottleneck of computational modeling is shifting from hardware performance towards that of software development, more specifically to the ability to develop more complex models and to verify and validate them in a timely and cost-efficient manner (Post and Votta, 2005). The importance of verification and validation is highlighted by the fact that even a trivial bug can have devastating consequences not only for users of the affected software but also for others who try to publish contradicting results (Miller, 2006).

Modular software can be (re)used with minimal modification and is advantageous not only for reducing development effort but also for verifying and validating a new program. For example the number of errors in software components that are reused without modification can be an order of magnitude lower than in components which are either developed from scratch or modified extensively before use (Thomas et al., 1997). The verification and validation (V&V) of a program consisting of several modules should start from V&V of each module separately before proceeding to combinations of modules and finally the entire program (Oberkampf and Trucano, 2002). Modules that have been verified and validated and are used without modification increase the confidence in the functionality of the larger program and decrease the effort required for final V&V.

Reusable software that does not depend on any specific type of data can be written by using, for example, generic programming (Musser and Stepanov, 1989). Waligora et al. (1995) reported that the use of object-oriented design and generics of the Ada programming language at the Flight Dy-

namics Division of NASA's Goddard Space Flight Center had increased software reuse by a factor of 3 and, in addition to other benefits, reduced the error rates and costs substantially. With C++, generic software can be developed without sacrificing computational performance through the use of compile-time template parameters for which the compiler can perform optimizations that would not be possible otherwise (e.g., Veldhuizen and Gannon, 1998; Stroustrup, 1999).

1.1 Model coupling

Generic and modular software is especially useful for developing complex computational models that couple together several different and possibly independently developed codes. From a software development point of view, code coupling can be defined as simply making the variables stored by different codes available to each other. In this sense even a model for the flow of incompressible, homogeneous and non-viscous fluid without external forcing

$$\frac{\partial v}{\partial t} = -v \cdot (\nabla v) - \nabla p; \nabla^2 p = -\nabla \cdot (v \cdot (\nabla v)),$$

where v is velocity and p is pressure, can be viewed as a coupled model as there are two equations that can be solved by different solvers. If a separate solver is written for each equation and both solvers are simulating the same volume with identical discretization, coupling will be only a matter of data exchange. In this work the term solver will be used when referring to any code/function/module/library which takes as input the data of a cell and its N neighbors and produces the next state of the cell (next step, iteration, temporal substep, etc.).

The methods of communicating data between solvers can vary widely depending on the available development effort, the programming language(s) involved and details of the specific codes. Probably the easiest coupling method to develop is to transfer data through the file system, i.e., at every step each solver writes the data needed by other solvers into a file and reads the data produced by other solvers from other files. This method is especially suitable as a first version of coupling when the codes have been written in different programming languages and use non-interoperable data structures.

Performance-wise a more optimal way to communicate between solvers in a coupled program is to use shared memory, as is done for example in Hill et al. (2004), Jöckel et al. (2005), Larson et al. (2005), Toth et al. (2005), Zhang and Parashar (2006) and Redler et al. (2010); nevertheless, this technique still has shortcomings. Perhaps the most important one is the fact that the data types used by solvers are not visible from the outside, thus making intrusive modifications (i.e., modifications to existing code or data structures) necessary in order to transfer data between solvers. The data must be converted to an intermediate format by the solver sending the data and subsequently converted to the internal format by the solver receiving the data. The probability of bugs is also increased as the code doing the end-to-end conversion is scattered between two different places and the compiler cannot perform static type checking for the final coupled program. These problems can be alleviated by, e.g., writing the conversion code in another language and outputting the final code of both submodels automatically (Eller et al., 2009). Interpolation between different grids and coordinate systems that many of the frameworks mentioned previously perform can also be viewed as part of the data transfer problem but is outside the scope of this work.

A distributed memory parallel program can require significant amounts of code for arranging the transfers of different variables between processes, for example, if the number of data required by some variable(s) changes as a function of both space and time. The problem is even harder if a program consists of several coupled models with different time stepping strategies and/or variables whose memory requirements change at run-time. Furthermore, modifying an existing time stepping strategy or adding another model into the program can require substantial changes to existing code in order to accommodate additional model variables and/or temporal substeps.

1.2 Generic simulation cell method

A generic simulation cell class is presented that provides an abstraction for the storage of simulation variables and the transfer of variables between processes in a distributed memory parallel program. Each variable to be stored in the generic cell class is given as a template parameter to the class. The variables of each cell instance are grouped together in memory; therefore, if several cell instances are stored contiguously in memory (e.g., in an std::vector) a variable will be interleaved with other variables in memory (see Sect. 8 for a discussion on how this might affect application performance). The type of each variable is not restricted in any way by the cell class or solvers developed using this abstraction, enabling generic programming in simulation development from the top down to a very low level. By using variadic templates of the 2011 version of the C++ standard (C++11), the total number of variables is only limited by the compiler implementation. A minimum of 1024 template arguments is recommended by C++11 (see, e.g., Annex B in Du Toit, 2012) and the cell class presented here can itself also be used as a variable thereby grouping related variables together and reducing the total number of template arguments given to the cell class.

By using the generic cell abstraction, it is possible to develop distributed memory parallel computational models in a way that easily allows one to combine an arbitrary number of separate models without modifying existing code. This is demonstrated by combining parallel models for Conway's Game of Life (GoL) (Gardner, 1970), scalar advection and Lagrangian transport of particles in an external velocity field. In order to keep the presented programs succinct, combining

computational models is defined here as running each model on the same grid structure with identical domain decomposition across processes. This is not mandatory for the generic cell approach and, for example, the case of different domain decomposition of submodels is discussed in Sect. 4.

Section 2 introduces the generic simulation cell class concept via a serial implementation and Sect. 3 extends it to distributed memory parallel programs. Section 4 shows that it is possible to combine three different computational models without modifying existing code by using the generic simulation cell method. Section 6 shows that the generic cell implementation developed here does not seem to have an adverse effect on either serial or parallel computational performance. The code is available at https://github.com/nasailja/gensimcell for everyone to use, study, modify and redistribute; users are kindly requested to cite this work. The relative paths to source code files given in the rest of the text refer to the version of the generic simulation cell tagged as 1.0 in the git repository and is available at https://github.com/nasailja/gensimcell/tree/1.0/. The presented generic simulation cell class requires a standard C++11 compiler and parallel functionality additionally requires a C implementation of Message Passing Interface (MPI) and the header-only Boost libraries MPL, Tribool and TTI.

2 Serial implementation

Lines 1–30 in Listing 1 show a serial implementation of the generic simulation cell class that does not provide support for MPI applications, is not const-correct and does not hide implementation details from the user but is otherwise complete. The cell class takes as input an arbitrary number of template parameters that correspond to variables to be stored in the cell. Variables have to only define their type through the name data_type. When the cell class is given one variable as a template argument the code on lines 3–11 is used. The variable given to the cell class as a template parameter is stored as a member of the cell class on line 4 and access to it is provided by the cell's [] operator overloaded for the variable's class on lines 6–10[1]. When given multiple variables as template arguments the code on lines 13–30 is used which similarly stores the first variable as a private member and provides access to it via the [] operator. Additionally the cell class derives from itself with one less variable on line 19. This recursion is stopped by eventually inheriting the one variable version of the cell class. Access to the private data members representing all variables are provided by the respective [] operators which are made available outside of the cell class on line 23. The memory layout of variables in an instance of the cell class depends on the compiler implementation and can include, for example, padding between variables given as consecutive template parameters. This also

```
1  template <class... Variables> struct Cell;
2
3  template <class Variable> struct Cell<Variable> {
4    typename Variable::data_type data;
5
6    typename Variable::data_type& operator[](
7      const Variable&
8    ) {
9      return this->data;
10   }
11 };
12
13 template <
14   class Current_Variable,
15   class... Rest_Of_Variables
16 > struct Cell<
17   Current_Variable,
18   Rest_Of_Variables...
19 > : public Cell<Rest_Of_Variables...> {
20
21   typename Current_Variable::data_type data;
22
23   using Cell<Rest_Of_Variables...>::operator[];
24
25   typename Current_Variable::data_type& operator[](
26     const Current_Variable&
27   ) {
28     return this->data;
29   }
30 };
31
32 struct Mass_Density { using data_type = double; };
33 struct Momentum_Density { using data_type = double[3]; };
34 struct Total_Energy_Density { using data_type = double; };
35 using HD_Conservative = Cell<
36   Mass_Density, Momentum_Density, Total_Energy_Density
37 >;
38
39 struct HD_State { using data_type = HD_Conservative; };
40 struct HD_Flux { using data_type = HD_Conservative; };
41 using Cell_T = Cell<HD_State, HD_Flux>;
42
43 int main() {
44   Cell_T cell;
45   cell[HD_Flux()][Mass_Density()]
46     = cell[HD_State()][Momentum_Density()][0];
47 }
```

Listing 1. Brief serial implementation of the generic simulation cell class (lines 1–30) and an example definition of a cell type used in a hydrodynamic simulation (lines 32–47).

applies to variables stored in ordinary structures, and in both cases if, for example, several values are to be stored contiguously in memory, a container such as std::array or std::vector guaranteeing this should be used.

Lines 32–47 in Listing 1 define a cell type for a hydrodynamic simulation and a program assigning a value from one variable of a cell to another variable of the same cell. The variables of hydrodynamic equations are defined on lines 32–34 and a cell type consisting of those variables is defined on lines 35–37. In order to store both the current state of the simulation and the change in state variables from one time step to the next, two variables are defined on lines 39 and 40 representing the current state and fluxes into and out of each cell. The final type stored in each cell of the simulation grid consists of the state and flux variables defined on lines 39–41.

[1] Future versions might implement a multi-argument () operator returning a tuple of references to several variables' data

3 Parallel implementation

In a parallel computational model, variables in neighboring cells must be transferred between processes in order to calculate the solution at the next time step or iteration. On the other hand it might not be necessary to transfer all variables in each communication as one solver could be using higher order time stepping than others and require more iterations for each time step. Furthermore, for example, when modeling an incompressible fluid the Poisson's equation for pressure must be solved at each time step, i.e., iterated in parallel until some norm of the residual becomes small enough, during which time other variables need not be transferred. Model variables can also be used for debugging and need not be transferred between processes by default.

Table 1 shows a summary of the application programming interface of the generic simulation cell class. It provides support for parallel programs via its get_mpi_datatype() member function which returns the information required to transfer one or more variables stored in the cell via a library implementing the MPI standard. An implementation of MPI is not required to use the generic simulation cell class, in that case (if the preprocessor macro MPI_VERSION is not defined when compiling) support for all MPI functionality will not be enabled and the class will behave as shown in Sect. 2. Transferring variables of a standard type (e.g., float, long int) or a container (of container of, etc.) of standard types (array, vector, tuple) is supported out of the box. The transfer of one or more variables can be switched on or off via a function overloaded for each variable on a cell-by-cell basis or for all cells of a particular cell type at once. This allows the communication strategy of a program to be formalized by explicitly stating which variables in what part of the simulated volume must be transferred between processes at different stages of execution for the correct functionality of each solver and the entire program.

The transfer of variables in all instances of a particular type of cell can be toggled with its set_transfer_all() function and the set_transfer() member can be used to toggle the transfer of variables in each instance of a cell type separately. The former function takes as arguments a boost::tribool value and the affected variables. If the tri-boolean value is determined (true of false) then all instances will behave identically for the affected variables, otherwise (a value of boost::indeterminate) the decision to transfer the affected variables is controlled on a cell-by-cell basis by the latter function. The functions are implemented recursively using variadic templates in order to allow the user to switch on/off the transfer of arbitrary combinations of variables in the same function call. The get_mpi_datatype() member function iterates through all variables stored in the cell at compile time and only adds variables which should be transferred to the final MPI_Datatype at run-time.

Listing 2 shows a parallel implementation of Conway's GoL using the generic simulation cell class. To simplify the implementation, the game grid is 1-dimensional, periodic in that dimension and with one cell per process. A version of this example with console output is available at https://github.com/nasailja/gensimcell/blob/1.0/examples/game_of_life/parallel/no_dccrg.cpp. When run with 11 processes the program prints as follows:

```
#include <tuple>
#include <vector>
#include <mpi.h>
#include <gensimcell.hpp>

struct Is_Alive { using data_type = bool; };
struct Live_Neighbors { using data_type = int; };
using Cell_T = gensimcell::Cell<
  gensimcell::Optional_Transfer, Is_Alive, Live_Neighbors
>;

int main(int argc, char* argv[]) {
  constexpr Is_Alive is_alive{};
  constexpr Live_Neighbors live_neighbors{};

  MPI_Init(&argc, &argv);
  int rank = 0, comm_size = 0;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

  Cell_T cell, neg_neigh, pos_neigh;
  cell[is_alive] = true; cell[live_neighbors] = 0;
  if (rank == 0) cell[is_alive] = false;

  Cell_T::set_transfer_all(true, is_alive);
  for (size_t turn = 0; turn < 10; turn++) {
    std::tuple<void*, int, MPI_Datatype>
      cell_info = cell.get_mpi_datatype(),
      neg_info = neg_neigh.get_mpi_datatype(),
      pos_info = pos_neigh.get_mpi_datatype();

    using std::get;
    MPI_Request neg_send, pos_send, neg_recv, pos_recv;
    MPI_Irecv(
      get<0>(neg_info), get<1>(neg_info), get<2>(neg_info),
      int(unsigned(rank + comm_size - 1) % comm_size),
      int(unsigned(rank + comm_size - 1) % comm_size),
      MPI_COMM_WORLD, &neg_recv
    );
    MPI_Irecv(
      get<0>(pos_info), get<1>(pos_info), get<2>(pos_info),
      int(unsigned(rank + 1) % comm_size),
      int(unsigned(rank + 1) % comm_size),
      MPI_COMM_WORLD, &pos_recv
    );
    MPI_Isend(
      get<0>(cell_info), get<1>(cell_info), get<2>(cell_info),
      int(unsigned(rank + comm_size - 1) % comm_size), rank,
      MPI_COMM_WORLD, &neg_send
    );
    MPI_Isend(
      get<0>(cell_info), get<1>(cell_info), get<2>(cell_info),
      int(unsigned(rank + 1) % comm_size), rank,
      MPI_COMM_WORLD, &pos_send
    );

    MPI_Wait(&neg_recv, MPI_STATUS_IGNORE);
    MPI_Wait(&pos_recv, MPI_STATUS_IGNORE);
    if (neg_neigh[is_alive]) cell[live_neighbors]++;
    if (pos_neigh[is_alive]) cell[live_neighbors]++;
    MPI_Wait(&neg_send, MPI_STATUS_IGNORE);
    MPI_Wait(&pos_send, MPI_STATUS_IGNORE);

    if (cell[live_neighbors] == 2) cell[is_alive] = true;
    else cell[is_alive] = false;
    cell[live_neighbors] = 0;
  }
  MPI_Finalize();
}
```

Listing 2. Parallel implementation of Conway's Game of Life using the generic simulation cell class.

```
.0000000000
0.00000000.
.0.000000.0
0.0.0000.0.
.0.0.00.0.0
0.0.0..0.0.
.0.0....0.0
0.0......0.
.0........0
0..........
..........
```

Lines 6–10 define the model's variables and the cell type used in the model grid. The first parameter given to the cell class on line 9 is the transfer policy of that cell type with three possibilities: (1) all variables are always transferred, (2) none of the variables are transferred and (3) transfer of each variable can be switched on and off as described previously. In the first two cases, all variables of a cell instance are laid out contiguously in memory, while in the last case a boolean value is added for each variable that records whether the variable should be transferred from that cell instance. Lines 13 and 14 provide a shorthand notation for referring to simulation variables. Lines 16–23 initialize MPI and the game. The [] operator is used to obtain a reference to variables' data, e.g., on line 22. Line 25 switches on the transfer of the Is_Alive variable whose information will now be returned by each cells' get_mpi_datatype() method on lines 28–30. Lines 27–58 of the time stepping loop transfer variables' data based on the MPI transfer information provided by instances of the generic simulation cell class. In this case only one variable is transferred by each cell hence all cells return the address of that variable (e.g., line 4 in Listing 1), a count of 1 and the equivalent MPI data type MPI_CXX_BOOL.

Listing 3 shows a program otherwise identical to the one in Listing 2 but which does not use the generic simulation cell class. Calls to the MPI library (Init, Isend, Irecv, etc.) are identical in both programs but there is also a fundamental difference: in Listing 3 the names of variables used in the program cannot be changed without affecting all parts of the program in which those variables are used, whereas in Listing 2 only lines 13 and 14 of the main function would be affected. This holds true even if the solver logic is separated into a function as shown in Listing 4. If the names of the one or more simulation variables were changed the function on lines 1–9 would not have to be changed, instead only the function call on lines 18–22 would be affected. In contrast all variable names in the function on lines 11–16 would have to be changed to match the names defined elsewhere. It should be noted that the serial functionality of generic cell class is available for std::tuple in the newest C++ standard approved on 2014, but it requires a bit of extra code and is more verbose to use (e.g., std::get<Mass_Density>(get<HD_Flux>(cell)) instead of line 45 in Listing 1) than the generic cell class presented here. Also, support for distributed memory parallel programs, which makes up bulk of the code of generic simulation cell class, is not available.

```
1  #include <tuple>
2  #include <vector>
3  #include <mpi.h>
4
5  struct Cell_T {
6    bool is_alive; int live_neighbors;
7    std::tuple<void*, int, MPI_Datatype> get_mpi_datatype() {
8      return std::make_tuple(&this->is_alive, 1, MPI_CXX_BOOL);
9    }
10 };
11
12 int main(int argc, char* argv[]) {
13   MPI_Init(&argc, &argv);
14   int rank = 0, comm_size = 0;
15   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
16   MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
17
18   Cell_T cell, neg_neigh, pos_neigh;
19
20   cell.is_alive = true; cell.live_neighbors = 0;
21   if (rank == 0) cell.is_alive = false;
22
23   for (size_t turn = 0; turn < 10; turn++) {
24     ...
25
26     MPI_Wait(&neg_recv, MPI_STATUS_IGNORE);
27     MPI_Wait(&pos_recv, MPI_STATUS_IGNORE);
28     if (neg_neigh.is_alive) cell.live_neighbors++;
29     if (pos_neigh.is_alive) cell.live_neighbors++;
30     MPI_Wait(&neg_send, MPI_STATUS_IGNORE);
31     MPI_Wait(&pos_send, MPI_STATUS_IGNORE);
32
33     if (cell.live_neighbors == 2) cell.is_alive = true;
34     else cell.is_alive = false;
35     cell.live_neighbors = 0;
36   }
37   MPI_Finalize();
38 }
```

Listing 3. Parallel implementation of Conway's Game of Life not using the generic simulation cell class. Line 24 marked with ... is identical to lines 27–55 in Listing 2.

Subsequent examples use the DCCRG library (Honkonen et al., 2013) for abstracting away the logic of transferring neighboring cells' data between processes, i.e., lines 27–58 and 61–62 in Listing 2. DCCRG queries the data to be transferred from each cells' get_mpi_datatype() function as needed, i.e., when a cell considers the cell of another process as a neighbor or vice versa, and passes on that information to the MPI transfer functions.

Listing 5 shows the variables and time stepping loop of a parallel advection program using the generic simulation cell class and DCCRG library, the full program is available at https://github.com/nasailja/gensimcell/tree/1.0/examples/advection/parallel/main.cpp. The solver function(s) uses a first order donor cell algorithm and are implemented similarly to lines 1–9 in Listing 4, i.e., each function takes the required data as arguments and as template arguments the variables to use when calculating the solution. Internally, the functions call the [] operator of each cell with instances of the variables given as template arguments to access the required data. Computation is overlapped with communication by solving cells without neighbors on other processes (lines 13–18), while data transfers are ongoing (lines 11 and 20).

```
1  template<class IA, class LN, class C> void solve(
2    C& cell, const C& neg, const C& pos
3  ) {
4    constexpr IA is_alive{};
5    constexpr LN live_neighbors{};
6
7    if (neg[is_alive]) cell[live_neighbors]++;
8    if (pos[is_alive]) cell[live_neighbors]++;
9  }
10
11 template<class C> void solve_no_gensimcell(
12   C& cell, const C& neg, const C& pos
13 ) {
14   if (neg.is_alive) cell.live_neighbors++;
15   if (pos.is_alive) cell.live_neighbors++;
16 }
17
18 solve<
19   Is_Alive, Live_Neighbors
20 >(
21   cell, neg_neigh, pos_neigh
22 );
23
24 solve_no_gensimcell(
25   cell, neg_neigh, pos_neigh
26 );
```

Listing 4. Functions for calculating the number of live neighbors for the program shown in Listing 2 on lines 1–9 and the program in Listing 3 on lines 11–16. Lines 18–26 show the function calls as they would appear in the respective main programs.

4 Combination of three simulations

Parallel models implemented using the generic simulation cell class can be combined without modification into one model by incorporating relevant parts from the main.cpp file of each model. This is enabled by the use of distinct types, or metavariables, for describing simulation variables and for accessing their data as well as by allowing one to easily switch on and off the transfer of variables' data between processes in a distributed memory parallel program. The time stepping loop of a program playing Conway's GoL, solving the advection equation and propagating particles in an external velocity field is shown in Listing 6 and the full program is available at https://github.com/nasailja/gensimcell/blob/1.0/examples/combined/parallel.cpp. Separate namespaces are used for the variables and solvers of each submodel as, e.g., two have a variable named Velocity and all have a function named solve. Variables of the parallel particle propagation model are shown on lines 1–11. In order not to lose particles between processes each particle that moves from one cell to another in the simulated volume is moved from the originating cell's internal particle list (lines 7–8) to its external particle list which includes the receiving cell's id (lines 9–11) and later incorporated into the receiving cell's internal particle list. For this reason the particle propagator solves the outer cells of a simulation first (lines 14–15) so that the number of particles moving between cells on different processes can be updated (lines 17–19) while the inner simulation cells are solved (lines 21–22). After the number of incoming particle data is known (line 24), space for that data is allocated (line 25) and eventually received (lines 31–33 and 41). Lines 35–59 solve the parallel GoL and advection problems while finishing up the particle solution for that time step.

```
1  struct Density { using data_type = double; };
2  struct Density_Flux { using data_type = double; };
3  struct Velocity {
4    using data_type = std::array<double, 2>; };
5  ...
6  while (simulation_time <= M_PI) {
7
8    Cell::set_transfer_all(true,
9      advection::Density(), advection::Velocity()
10   );
11   grid.start_remote_neighbor_copy_updates();
12
13   advection::solve<
14     Cell,
15     advection::Density,
16     advection::Density_Flux,
17     advection::Velocity
18   >(time_step, inner_cells, grid);
19
20   grid.wait_remote_neighbor_copy_update_receives();
21
22   advection::solve<
23     Cell,
24     advection::Density,
25     advection::Density_Flux,
26     advection::Velocity
27   >(time_step, outer_cells, grid);
28
29   advection::apply_solution<
30     Cell,
31     advection::Density,
32     advection::Density_Flux
33   >(inner_cells, grid);
34
35   grid.wait_remote_neighbor_copy_update_sends();
36   Cell::set_transfer_all(false,
37     advection::Density(), advection::Velocity()
38   );
39
40   advection::apply_solution<
41     Cell,
42     advection::Density,
43     advection::Density_Flux
44   >(outer_cells, grid);
45
46   simulation_time += time_step;
47 }
```

Listing 5. Variables and time stepping loop of a parallel advection program that uses the generic simulation cell class and DCCRG (Honkonen et al., 2013).

All submodels of the combined model run in the same discretized volume and with identical domain decomposition. This is not mandatory though as the cell id list given to each solver need not be identical, but in this case the memory for all variables in all cells is always allocated when using variables shown, e.g., in Listing 2. If submodels always run in non-overlapping or slightly overlapping regions of the simulated volume, and/or a method other than domain decomposition is used for parallelization, the memory required for the variables can be allocated at run-time in regions/processes

```
1  struct Number_Of_Internal_Particles {
2    using data_type = unsigned long long int; };
3  struct Number_Of_External_Particles {
4    using data_type = unsigned long long int; };
5  struct Velocity {
6    using data_type = std::array<double, 2>; };
7  struct Internal_Particles { using data_type
8    = std::vector<std::array<double, 3>>; };
9  struct External_Particles { using data_type
10   = std::vector<std::pair<
11   std::array<double, 3>, unsigned long long int>>;};
12 ...
13 while (simulation_time <= M_PI) {
14   particle::solve<...>(
15     time_step, outer_cells, grid)
16
17   Cell::set_transfer_all(true,
18     particle::Number_Of_External_Particles());
19   grid.start_remote_neighbor_copy_updates();
20
21   particle::solve<...>(
22     time_step, inner_cells, grid)
23
24   grid.wait_remote_neighbor_copy_update_receives();
25   particle::resize_receiving_containers<...>(grid);
26
27   grid.wait_remote_neighbor_copy_update_sends();
28
29   Cell::set_transfer_all(true, gol::Is_Alive());
30   ...
31   Cell::set_transfer_all(true,particle::Velocity(),
32     particle::External_Particles());
33   grid.start_remote_neighbor_copy_updates();
34
35   gol::solve<...>(inner_cells, grid);
36   advection::solve<...>(
37     time_step, inner_cells, grid)
38   particle::incorporate_external_particles<...>(
39     inner_cells, grid);
40
41   grid.wait_remote_neighbor_copy_update_receives();
42
43   gol::solve<...>(outer_cells, grid);
44   advection::solve<...>(
45     time_step, outer_cells, grid)
46   gol::apply_solution<...>(inner_cells, grid);
47   advection::apply_solution<...>(
48     inner_cells, grid);
49   particle::incorporate_external_particles<...>(
50     outer_cells, grid);
51   particle::remove_external_particles<...>(
52     inner_cells, grid);
53
54   grid.wait_remote_neighbor_copy_update_sends();
55
56   gol::apply_solution<...>(outer_cells, grid);
57   advection::apply_solution<...>(outer_cells, grid);
58   particle::remove_external_particles<...>(
59     outer_cells, grid);
60
61   simulation_time += time_step;
62 }
```

Listing 6. Illustration of the time stepping loop of a parallel program playing Conway's Game of Life, solving the advection equation and propagating particles in an external velocity field. Code from parallel advection simulation (Listing 5) did not have to be changed. Variables of the particle propagation model are also shown.

Table 1. Summary of application programming interface of generic simulation cell class. Listed functions are members of the cell class unless noted otherwise.

Code	Explanation	Usage example on line # of Listing 2
template <class TP, class... Variables>	Transfer policy and arbitrary number of variables to be stored in a cell type given as template arguments	9
operator[] (const Variable& v)	Returns a reference to given variable's data	22
get_mpi_datatype()	Returns MPI transfer information for variables selected for transfer	28
set_transfer(const bool b, const Variables&... vs)	Switches on or off transfer of given variables in a cell	not used
set_transfer_all(const boost::tribool b, const Variables&... vs)	Static class function for switching the transfer of given variables in all cell instances on, off or to be decided on a cell-by-cell basis by set_transfer()	25
Member operators $+=$, $-=$, $*=$, $/=$ and free operators $+$, $-$, $*$ and $/$	Operations for two cells of the same type or a cell and standard types, the operator is applied to each variable stored in given cell(s)	not used

where the variables are used. This can potentially be easily accomplished, for example, by wrapping the type of each variable in the boost::optional[2] type.

5 Coupling models that use generic simulation cell method

In the combined model shown in the previous section, the submodels cannot affect each other as they all use different variables and are thus unable to modify each other's data. In order to couple any two or more submodels new code must be written or existing code must be modified. The complexity of this task depends solely on the nature of the coupling. In simple cases where the variables used by one solver are only

[2]http://www.boost.org/doc/libs/1_57_0/libs/optional/doc/html/index.html

```
1  if (std::fmod(simulation_time, 1) < 0.5) {
2    particle::solve<
3      Cell,
4      particle::Number_Of_Internal_Particles,
5      particle::Number_Of_External_Particles,
6      advection::Velocity, // clock-wise
7      particle::Internal_Particles,
8      particle::External_Particles
9    >(time_step, outer_cells, grid)
10 } else {
11   particle::solve<
12     Cell,
13     particle::Number_Of_Internal_Particles,
14     particle::Number_Of_External_Particles,
15     particle::Velocity, // counter clock-wise
16     particle::Internal_Particles,
17     particle::External_Particles
18   >(time_step, outer_cells, grid)
19 }
```

Listing 7. Example of one way coupling between the parallel advection and particle propagation models. The clock-wise rotating velocity field of the advection model (line 6) is periodically used by the particle propagation model instead of the counter clock-wise rotating velocity field of the particle propagation model (line 15).

switched to variables of another solver, only the template parameters given to the solver have to be switched.

Listing 7 shows an example of a one way coupling of the parallel particle propagation model with the advection model by periodically using the velocity field of the advection model in the particle propagation model. A similar approach could be relevant, e.g., in modeling atmospheric transport of volcanic ash in velocity fields of a weather prediction model (Kerminen et al., 2011). On line 6 the particle solver is called with the velocity field of the advection model, while on line 15 the particle model's regular velocity field is used. More complex cases of coupling, which require, e.g., additional variables or transferring variable data between processes, are also simple to accomplish from the software development point of view. Additional variables can be freely inserted into the generic cell class and used by new couplers without affecting other submodels and the transfer of new variables can be activated by inserting calls to, e.g., the generic cell's set_transfer_all() function as needed.

6 Effect on serial and parallel performance

In order to be usable in practice, the generic cell class should not slow down a computational model too much and ideally neither serial nor parallel performance should be affected when compared to hand-written code with identical functionality. I test this using two programs: a serial GoL model and a parallel particle propagation model. The tests are conducted on a four core 2.6 GHz Intel Core i7 CPU with 256 kB L2 cache per core, 16 GB of 1600 MHz DDR3 RAM and the following software: OS X 10.9.2, GCC 4.8.4_0 from MacPorts, Open MPI 1.8.4, Boost 1.56.0 and dccrg commit 7d5580a30 dated 12 January 2014 from the c++11 branch at https://gitorious.org/dccrg/dccrg. The programs are compiled with –O3 –std=c++0x. Where available/supported the options –march=native and –mtune=native were found to increase program speed by about 0.2 % with GCC on another machine and 3 % with Clang on the hardware described above.

Table 2. Run-times of serial GoL programs using different cell types and memory layouts for their variables compiled with GCC 4.8.4.

Cell type	Memory layout	Run-time(s)
generic	bool first	1.422
generic	int first	1.601
struct	bool first	1.424
struct	int first	1.603
struct	bool first, both aligned to 8 B	1.552
struct	int first, both aligned to 8 B	1.522

6.1 Serial performance

Serial performance of the generic cell is tested by playing GoL for 30 000 steps on a 100 by 100 grid with periodic boundaries and allocated at compile time. Performance is compared against an implementation using `struct { bool; int; };` as the cell type. Both implementations are available in the directory https://github.com/nasailja/gensimcell/tree/1.0/tests/serial/game_of_life/. Each timing is obtained by executing five runs, discarding the slowest and fastest runs and averaging the remaining three runs. As shown in Table 2 serial performance is not affected by the generic cell class, but the memory layout of variables regardless of the cell type used affects performance by over 10 %. Column two specifies whether is_alive or live_neighbors is stored at a lower memory address in each cell. By default the memory alignment of the variables is implementation defined but on the last two rows of Table 2 alignas(8) is used to obtain 8 byte alignment for both variables. The order of the variables in memory in a generic cell consisting of more than one variable is not defined by the standard. On the tested system the variables are laid out in memory by GCC in reverse order with respect to the order of the template arguments. Other compilers do not show as large a difference between different ordering of variables: with ICC 14.0.2 all run-times are about 6 s using either –O3 or –fast (alignas is not yet supported) and with Clang 3.4 from MacPorts all run-times are about 3.5 s (about 3.6 s using alignas).

6.2 Parallel performance

Parallel performance of the generic cell class is evaluated with a particle propagation test which uses more complex variable types than the GoL test in order to emphasize the computational cost of setting up MPI transfer information in the generic cell class and a manually written reference cell class. Both implementations are available in the directory https://github.com/nasailja/gensimcell/tree/1.0/tests/parallel/particle_propagation/. Parallel tests are compiled with GCC and are run using three processes and the final time is the average of the times reported by each process. Similarly to the serial case, each test is executed five times, outliers are discarded and the final result averaged over the remaining three runs. The tests are run on a 20^3 grid without periodic boundaries and RANDOM load balancing is used to emphasize the cost of MPI transfers. The run-time of the generic cell class version is 2.2 s and the reference program is 3.1 s. The extra time in reference program is spent in MPI regions but more detailed profiling was not done as the point is only to show that the generic simulation cell class does not slow down the parallel program. The output files of the different versions are bit identical if the same number of processes is used. When using recursive coordinate bisection load balancing, the run-times of both are about 0.5 s with the reference program being 10 % slower. A similar result is expected for a larger number of processes as the bottleneck will likely be in the actual transfer of data instead of the logic for setting up the transfers.

7 Converting existing software

Existing software can be gradually converted to use a generic cell syntax but the details depend heavily on the modularity of said software and especially on the way data in transferred between processes. If a grid library decides what to transfer and where and the cells provide this information via an MPI data type, conversion will likely require only small changes.

Listing 8 shows an example of converting a Conway's GoL program using cell-based storage (after Listing 4 of Honkonen et al., 2013) to the application programming interface used by the generic cell class. In this case the underlying grid library handles data transfers between processes so the only additions required are empty classes for denoting simulation variables and the corresponding [] operators for accessing the variables' data. With these additions the program can be converted step-by-step to use the generic cell class API and once completed the cell implementation shown in Listing 8 can be swapped with the generic cell.

```
1  struct game_of_life_cell {
2    int data[2];
3
4    std::tuple<
5      void*,
6      int,
7      MPI_Datatype
8    > get_mpi_datatype() const {
9      return std::make_tuple(
10       (void*) &(this->data[0]),
11       1,
12       MPI_INT
13     );
14   }
15 };
16
17 struct Is_Alive {};
18 struct Live_Neighbors {};
19
20 struct game_of_life_cell_compat {
21   ...
22   int& operator[](const Is_Alive&) {
23     return this->data[0];
24   }
25
26   int& operator[](const Live_Neighbors&) {
27     return this->data[1];
28   }
29 };
```

Listing 8. An example of converting existing software to use an application programming interface (API) identical to the generic cell class. API conversion consists of adding empty classes for denoting simulation variables on lines 17 and 18, and adding [] operators for accessing the variables' data on lines 22–28. Line 21 is identical to lines 2–14.

8 Discussion

The presented generic simulation cell method has several advantages over implementations using less generic programming methods:

1. The changes required for combining and coupling models that use the generic simulation cell class are minimal and in the presented examples no changes to existing code are required for combining models. This is advantageous for program development as submodels can be tested and verified independently and also subsequently used without modification which decreases the possibility of bugs and increases confidence in the correct functioning of the larger program.

2. The generic simulation cell method enables zero-copy code coupling as an intermediate representation for model variables is not necessary due to the data types of simulation variables being visible outside of each model. Thus, if coupled models use a compatible representation for data, such as IEEE floating point, the vari-

ables of one model will be used directly by another one without the first model having to export the data to an intermediate format. This again decreases the chance for bugs by reducing the required development effort and by allowing the compiler to perform type checking for the entire program and warn in cases of, e.g., undefined behavior (Wang et al., 2012).

3. Arguably code readability is also improved by making simulation variables separate classes and making models a composition of such variables. Shorthand notation for accessing variables' data is also possible if the reduction in verbosity is deemed acceptable:

```
constexpr Mass_Density Rho{};
constexpr Velocity V{};
cell_data[Rho] = ...;
cell_data[V][0] = ...;
cell_data[V][1] = ...;
...
```

For many intents and purposes the presented cell class acts identically to the standard heterogeneous container std::tuple (see also Sect. 3 on using tuple as a substitute in serial code) while providing additional syntactical sugar for serial programs and helpful functionality for distributed memory parallel programs.

For example the question of memory layout of simulation variables, whether each simulation variable should be stored contiguously in memory or interleaved with other variables at the same location in the simulated volume, also applies not only to using standard C++ containers but also to other programming languages as well. The key factor in this case seems to be the locality principle (Denning, 2005), i.e., that all hierarchies of computer (registers, caches, etc.) storage are reused as efficiently as possible while processing. For simulations modeling systems of multiple coupled equations it could well be that storing related variables, representing the same location of the simulated volume, contiguously in memory leads to faster execution than storing each variable separately from others. This is because, e.g., the smallest unit a CPU cache operates on is of the order of 100 bytes[3] so fetching variables of one neighbor cell can lead to many more reads from memory if each variable is stored in a separate array instead of being stored close to the neighbor's other variables. It should be quite simple to add the functionality of the cell class presented here to an existing grid library that would store all variables in separate contiguous arrays but this might not result in faster program execution and would violate at least rules 5 and 33 of Sutter and Alexandrescu (2011), namely "give one entity one cohesive responsibility" and "prefer minimal classes to monolithic classes". If such functionality is absolutely necessary one could store e.g., one DCCRG grid (which itself stores only one type such as double in each of its own cells) in each variable given to the generic simulation cell class.

Other methods of speeding up a program, such as vectorization and threading, do not seem to be affected by using the generic simulation cell class. For example the run-time of a threaded version of the combined parallel program (available in https://github.com/nasailja/gensimcell/blob/1.0/examples/combined/parallel_async.cpp), which uses std::async to (potentially) launch each solver in a new thread at each time step, is reduced to less than 75 % of the original program on the system described in Sect. 6.1 when using a 100 by 100 grid without I/O. As different solvers cannot access each other's data they can be run simultaneously in different threads. If submodels are coupled (e.g., in Listing 7) standard concurrency mechanisms can be used such as std::atomic or std::mutex. Access to entire simulation cells can be serialized with a mutex or it can guard a group of related variables (e.g., one for each of lines 39 and 40 in Listing 1) or single variables.

The possibility of using a generic simulation cell approach in the traditional high-performance language of choice – Fortran – seems unlikely as it currently lacks support for compile-time generic programming (McCormack, 2005). For example, a recently presented computational fluid dynamics package implemented in Fortran, using an object-oriented approach and following good software development practices (Zaghi, 2014), uses hard-coded names for variables throughout the application. Thus, if the names of any variables had to be changed for some reason, e.g., coupling to another model using identical variable names, all code using those variables would have to be modified and tested to make sure no bugs have been introduced.

9 Conclusions

I present a generic simulation cell method which allows one to write generic and modular computational models without sacrificing serial or parallel performance or code readability. I show that by using this method it is possible to combine several computational models without modifying any existing code and only write new code for coupling models. This is a significant advantage for model development which reduces the probability of bugs and eases development, testing and validation of computational models. Performance tests indicate that the effect of the presented generic simulation cell class on serial performance is negligible and parallel performance may even improve noticeably when compared to hand-written MPI logic.

The Supplement related to this article is available online at doi:10.5194/gmd-8-473-2015-supplement.

[3] Cache Hierarchy in http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

Acknowledgements. The author gratefully acknowledges Alex Glocer for insightful discussions and the NASA Postdoctoral Program for financial support.

Edited by: P. Jöckel

References

Denning, P. J.: The Locality Principle, Communications of the ACM, 48, 19–24, doi:10.1145/1070838.1070856, 2005.

Du Toit, S.: Working Draft, Standard for Programming Language C++, ISO/IEC, available at: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf (last access: 4 March 2015), 2012.

Eller, P., Singh, K., Sandu, A., Bowman, K., Henze, D. K., and Lee, M.: Implementation and evaluation of an array of chemical solvers in the Global Chemical Transport Model GEOS-Chem, Geosci. Model Dev., 2, 89–96, doi:10.5194/gmd-2-89-2009, 2009.

Gardner, M.: Mathematical Games, Sci. Am., 223, 120–123, doi:10.1038/scientificamerican1170-116, 1970.

Hill, C., DeLuca, C., Balaji, V., Suarez, M., and Silva, A. D.: The Architecture of the Earth System Modeling Framework, Comput. Sci. Eng., 6, 18–28, doi:10.1109/MCISE.2004.1255817, 2004.

Honkonen, I., von Alfthan, S., Sandroos, A., Janhunen, P., and Palmroth, M.: Parallel grid library for rapid and flexible simulation development, Comput. Phys. Commun., 184, 1297–1309, doi:10.1016/j.cpc.2012.12.017, 2013.

Jöckel, P., Sander, R., Kerkweg, A., Tost, H., and Lelieveld, J.: Technical Note: The Modular Earth Submodel System (MESSy) – a new approach towards Earth System Modeling, Atmos. Chem. Phys., 5, 433–444, doi:10.5194/acp-5-433-2005, 2005.

Kerminen, V.-M., Niemi, J. V., Timonen, H., Aurela, M., Frey, A., Carbone, S., Saarikoski, S., Teinilä, K., Hakkarainen, J., Tamminen, J., Vira, J., Prank, M., Sofiev, M., and Hillamo, R.: Characterization of a volcanic ash episode in southern Finland caused by the Grimsvötn eruption in Iceland in May 2011, Atmos. Chem. Phys., 11, 12227–12239, doi:10.5194/acp-11-12227-2011, 2011.

Larson, J., Jacob, R., and Ong, E.: The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models, Int. J. High Perform. C., 19, 277–292, doi:10.1177/1094342005056115, 2005.

McCormack, D.: Generic Programming in Fortran with Forpedo, SIGPLAN Fortran Forum, 24, 18–29, doi:10.1145/1080399.1080401, 2005.

Miller, G.: A Scientist's Nightmare: Software Problem Leads to Five Retractions, Science, 314, 1856–1857, doi:10.1126/science.314.5807.1856, 2006.

Musser, D. R. and Stepanov, A. A.: Generic programming, in: Symbolic and Algebraic Computation, edited by: Gianni, P., Vol. 358 of Lecture Notes in Computer Science, 13–25, Springer Berlin Heidelberg, doi:10.1007/3-540-51084-2_2, 1989.

Oberkampf, W. L. and Trucano, T. G.: Verification and validation in computational fluid dynamics, Prog. Aerosp. Sci., 38, 209–272, doi:10.1016/S0376-0421(02)00005-2, 2002.

Post, D. E. and Votta, L. G.: Computational science demands a new paradigm, Physics Today, 58, 35–41, doi:10.1063/1.1881898, 2005.

Redler, R., Valcke, S., and Ritzdorf, H.: OASIS4 – a coupling software for next generation earth system modelling, Geosci. Model Dev., 3, 87–104, doi:10.5194/gmd-3-87-2010, 2010.

Stroustrup, B.: Learning Standard C++ As a New Language, C/C++ Users J., 17, 43–54, 1999.

Sutter, H. and Alexandrescu, A.: C++ Coding Standards, C++ In-Depth Series, Addison-Wesley, available at: http://www.gotw.ca/publications/c++cs.htm (last access: 4 March 2015), 2011.

Thomas, W. M., Delis, A., and Basili, V. R.: An Analysis of Errors in a Reuse-oriented Development Environment, J. Syst. Softw., 38, 211–224, doi:10.1016/S0164-1212(96)00152-5, 1997.

Toth, G., Sokolov, I. V., Gombosi, T. I., Chesney, D. R., Clauer, C. R., De Zeeuw, D. L., Hansen, K. C., Kane, K. J., Manchester, W. B., Oehmke, R. C., Powell, K. G., Ridley, A. J., Roussev, I. I., Stout, Q. F., Volberg, O., Wolf, R. A., Sazykin, S., Chan, A., Yu, B., and Kota, J.: Space Weather Modeling Framework: A new tool for the space science community, J. Geophys. Res.-Space, 110, A12226, doi:10.1029/2005JA011126, 2005.

Veldhuizen, T. L. and Gannon, D.: Active Libraries: Rethinking the roles of compilers and libraries, in: In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO-98, SIAM Press, 1998.

Waligora, S., Bailey, J., and Stark, M.: Impact Of Ada And Object-Oriented Design In The Flight Dynamics Division At Goddard Space Flight Center, Tech. rep., National Aeronautics and Space Administration, Goddard Space Flight Center, 1995.

Wang, X., Chen, H., Cheung, A., Jia, Z., Zeldovich, N., and Kaashoek, M. F.: Undefined Behavior: What Happened to My Code?, in: Proceedings of the Asia-Pacific Workshop on Systems, APSYS'12, 9:1–9:7, ACM, New York, NY, USA, doi:10.1145/2349896.2349905, 2012.

Zaghi, S.: OFF, Open source Finite volume Fluid dynamics code: A free, high-order solver based on parallel, modular, object-oriented Fortran API, Computer Physics Communications, 185, 2151–2194, doi:10.1016/j.cpc.2014.04.005, 2014.

Zhang, L. and Parashar, M.: Seine: A Dynamic Geometry-based Shared Space Interaction Framework for Parallel Scientific Applications, in: In Proceedings of High Performance Computing – HiPC 2004: 11th International Conference, 189–199, Springer LNCS, 2006.