Geosci. Model Dev. Discuss., 7, 4577–4602, 2014
www.geosci-model-dev-discuss.net/7/4577/2014/
doi:10.5194/gmdd-7-4577-2014
© Author(s) 2014. CC Attribution 3.0 License.

Geoscientific Model Development Discussions
Open Access

The 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

Received: 11 June 2014 – Accepted: 30 June 2014 – Published: 18 July 2014

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

Published by Copernicus Publications on behalf of the European Geosciences Union.

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 any modification of existing code. This is an advantage for the development and testing of computational modeling software as each submodel can be developed and tested independently and subsequently used without modification in a more complex coupled program. Support for parallel programming is also provided by allowing users to select which simulation variables to transfer between processes via a Message Passing Interface 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 presented here requires a C++ compiler that supports variadic templates which were standardized in 2011 (C++11). The code is available at: https://github.com/nasailja/gensimcell for everyone to use, study, modify and redistribute; those that do are kindly requested to 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 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 V & V'd 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 Flight Dynamics Division of NASA's Goddard Space Flight Center had increased sofware reuse by a factor of three 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 (Stroustrup, 1999).

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

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

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 is 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 filesystem, 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), but this technique still has shortcomings. Perhaps the most important one is the fact that the data types used by solvers are not visible to 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 in 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 amount 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. Futhermore, modifying an existing time stepping strategy or adding another model into the program can require substatial changes to existing code in order to accomodate additional model variables and/or temporal substeps.

1.1 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 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 all the way from the top down to a very low level. By using variadic templates of the 2011 version of the C++ standard, the total number of variables is only limited by the compiler implementation and a minimum of 1024 is recommended by C++11 (see e.g. Annex B in Du Toit, 2012).

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 any existing code. This is demonstrated by combining parallel models for Conway's Game of Life, 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 accross 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. Also the definition of modifying existing code excludes copying and pasting unmodified code into a new file.

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 any existing code by using the generic simulation cell method. Section 5 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 0.5 in the git repository and is available at: https://github.com/nasailja/gensimcell/tree/0.5/.

2 Serial implementation

Figure 1 shows a basic implementation of the generic simulation cell class that does not provide support for MPI applications and is not const-correct 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. Each varible only defines its type through the name data_type (e.g. lines 5 and 6 in Fig. 2) and is otherwise empty. When the cell class is given one variable as a template argument the code on lines 3..13 is used. The variable given to the cell class as a template parameter is stored as a private member of the cell class on line 5 and access to it is provided by the cell's [] operator overloaded for the variable's class on lines 8..12. When given multiple variables as template arguments the code on lines 15..33 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 from itself with one less variable on line 21. 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 to outside of the cell class on line 26. 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 applies to variables stored in “ordinary” structures and in both cases if, for example, several values must be stored contiguously in memory a container guaranteeing this should be used such as std::array or std::vector.

Figure 2 shows a complete serial implementation of Conway’s Game of Life (GoL) using the generic simulation cell class. A version of this example with console output is available at: examples/game_of_life/serial.cpp[1]. Lines 5..7 define the variables to be used in the model and the cell type to be used in the model grid. Lines 10..20 create the simulation grid and initialize the simulation with a pseudorandom initial condition. The time stepping loop spans lines 22..57. The [] operator is used to obtain a reference to the data of all variables e.g. on lines 17 and 42. Lines 26 and 34 provide a short hand name for the curret cell and its neighbors respectively. Using the generic cell class adds hardly any code compared to a traditional implementation (e.g. Fig. 4 in Honkonen et al., 2013) and allows the types of the variables used in the model to be defined outside of both the grid which stores the simulation variables and the solver functions which use the variables to calculate the solution.

Figure 3 shows excerpts from serial versions of programs modeling advection and particle propagation in prescribed velocity fields. The full examples are available at: examples/advection/serial.cpp[2] and examples/particle_propagation/serial.cpp[3]. The variables of both models are defined similarly to Fig. 2 and the [] operator is used to refer to the variables’ data in each cell.

[1]https://github.com/nasailja/gensimcell/blob/0.5/examples/game_of_life/serial.cpp

[2]https://github.com/nasailja/gensimcell/blob/0.5/examples/advection/serial.cpp

[3]https://github.com/nasailja/gensimcell/blob/0.5/examples/particle_propagation/serial.cpp

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. Or, 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. Several model variables can also be used for debugging and need not always be transferred between processes.

The generic cell class provides support for parallel programs via a get_mpi_datatype() member function which can be used to query what data should be transferred to/from a generic cell with MPI. 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. 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 solver and the entire program. The parallel code presented here is built on top of the dccrg library Honkonen et al. (2013) which handles the details of e.g. transferring the data of neighboring simulation cells accross process boundaries by calling the get_mpi_datatype() member function of each cell when needed.

Standard types whose size is known at compile-time (e.g. int, float, std::array<int, N>, but see Sect. 3 for the general case) can be transferred without extra code from the user. Functions are provided by the cell class for switching on or off the transfer of one or more variables in all instances of a particular type of cell (set_transfer_all()) and for switching on or off the transfers in each instance separately (set_transfer()). The former function takes as arguments a boost::tribool value and the affected variables. If the triboolean value is determined (true of false) then all instances behave identically for the affected variables, otherwise (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. When the cell class returns the MPI transfer information via its get_mpi_datatype() member function, all variables are iterated through at compile-time and only those that should be transferred are added at run-time to the final MPI_Datatype returned by the function.

Figure 4 shows excerpts from the parallel version of the GoL example implemented using the generic simulation cell class and the dccrg grid library. The cell type used by this version (line 7) is identical to the type used in the serial version on line 7 of Fig. 2. In the parallel version the for loops over cells and their neighbors inside the time stepping loop have been moved to separate functions called solve and apply_solution respectively. At each time step, before starting cell data transfers between processes at process boundaries (line 21), the transfer of required variables, in this case whether the cell is alive or not, is switched on in all cells (line 20). No additional code is required for the transfer logic in contrast to a program not using the generic cell class (e.g. lines 12 and 13 in Fig. 4 of Honkonen et al., 2013). The function solving the system for a given list of cells (called solve in the namespace gol), which in this case counts the number of life neighbors, is called on lines 22..26 with the cell class and variables to use internally given as template parameters. This makes the function accept cells consisting of arbitrary variables and also allows one to change the variables used by the function easily.

The strategy used in Fig. 4 for overlapping computation and communication is also used in the other parallel examples. After starting data transfers between the outer cells of different processes the solution is calculated in the inner cells. Inner cells are defined as cells that do not consider cells on other processes as neighbors and that are not considered as neighbors of cells on other processes. Outer cells are defined as cells other than the inner cells of a process. Once the data of other processes' outer cells

has arrived the solution is calculated in local outer cells. After this the solution can be applied to inner cells and when the data of outer cells has arrived to other processes the solution can also be applied to the outer cells.

Figure 5 shows the variables used in the parallel particle propagation example available at: examples/particle_propagation/parallel/[4]. The particles in each cell are stored as a vector of arrays, i.e. the dimensionality of particle coordinates is known at compile time but the number of particle in each cell is not. As MPI requires that the (maximum) amount of data to receive from another process is known in advance the number of particles in a cell must be transferred in a separate message before the particles themselves. Here the number of particles is stored as a separate variable named Number_Of_Particles. The transfer of variables with complex types, for example types whose size or memory layout changes at run time, is supported via the get_mpi_datatype() mechanism, i.e. such types must define a get_mpi_datatype() function which provides the required transfer information to the generic cell class. For example in Fig. 5 the variables themselves again only define the type of the variable on lines 19 and 64 while the types themselves contain the logic related to MPI transfer information. In order to be able to reliably move particles between cells on different processes, i.e. without creating duplicates or loosing particles, the particle propagator uses two collections of particles, one for particles that stay inside of the cell in which they are stored (lines 1..16) and another for particles which have moved outside of their cell (lines 22..61). The latter variable includes information about which cell a particles have moved to.

4 Combination of three simulations

The examples of parallel models shown in Sect. 3 can all be combined into one model without modifying any existing code by copying and pasting the relevant parts

[4]https://github.com/nasailja/gensimcell/tree/0.5/examples/particle_propagation/parallel/

from the main.cpp file of each model into a combined cpp file available at: examples/combined/parallel.cpp[5]. This is enabled by using a separate namespace for the solvers and variables of each submodel, as e.g. two of them use a variable with an identical name (Velocity) and all submodels include a function named solve. 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 simple variables shown e.g. in Figs. 2 or 5. If submodels always run in non-overlapping or slightly overlapping regions of the simulated volume, and/or with different domain decomposition, the memory required for the variables can be allocated at run time in regions/processes where the variables are used. This can potentially be accomplished easily by wrapping the type of each variable in the boost::optional[6] type, for example.

4.1 Coupling

In the combined model shown in 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 switched to variables of another solver, only the template parameters given to the solver have to be switched. The template parameters decide which variables a solver should use, i.e. the GoL solver (examples/game_of_life/parallel/gol_solve.hpp[7]) internally uses a template parameter Is_Alive_T to refer to a variable which records whether a cell is alive or not

[5]https://github.com/nasailja/gensimcell/blob/0.5/examples/combined/parallel.cpp

[6]http://www.boost.org/doc/libs/1_55_0/libs/optional/doc/html/index.html

[7]https://github.com/nasailja/gensimcell/blob/0.5/examples/game_of_life/parallel/gol_solve.hpp

and the actual variable used for that purpose (Is_Alive) is given to the solver function in the main program (examples/game_of_life/parallel/main.cpp[8]).

Figure 6 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. On line 6 the particle solver is called with the velocity field of the advection model as the velocity variable to use while on line 15 the particle model's regular velocity field is used. More complex cases of coupling, which require e.g. additional variables, 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 any other submodels.

5 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. 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 (installed from MacPorts where applicable): OS X 10.9.2, GCC 4.8.2_0, Open MPI 1.7.4, Boost 1.55.0_1 and dccrg commit 7d5580a30 dated 12 January 2014 from the c++11 branch at https://gitorious.org/dccrg/dccrg. The test programs are compiled with -O3 -std=c++0x.

5.1 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

[8]https://github.com/nasailja/gensimcell/blob/0.5/examples/game_of_life/parallel/main.cpp

type. Both implementations are available in the directory tests/serial/game_of_life[9]. Each timing is obtained by executing 5 runs, discarding the slowest and fastest runs and averaging the remaining 3 runs. As shown in Table 1 serial performance is not affected by the generic cell class, but the memory layout of variables regardless of the cell type used affects performance over 10 %. Column 2 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 1 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 differences 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).

5.2 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 tests/parallel/particle_propagation[10]. Parallel tests are run using 3 processes and the final time is the average of the times reported by each process. Similarly to the serial case each test is executed 5 times, outliers are discarded and the final result averaged over the remaining 3 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. Again there is an insignificant difference between the run times of both versions

[9]https://github.com/nasailja/gensimcell/tree/0.5/tests/serial/game_of_life/

[10]https://github.com/nasailja/gensimcell/tree/0.5/tests/parallel/particle_propagation/

as the run time for the generic cell class version is 2.86 s while the reference implementation runs in 2.92 s. 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 are also comparable but almost an order of magnitude lower (about 0.5 s). 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.

6 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 datatype, conversion will likely require only small changes.

Figure 7 shows an example of converting a Conway's Game of Life program using cell-based storage (after Fig. 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 complete the cell implementation shown in Fig. 7 can be swapped with the generic cell.

7 Discussion

The presented generic simulation cell method has several advantages over traditional implementations:

1. The changes requred for combining and coupling models 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 confidense 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 variables of one model can 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 composing models from a set of such variables. Shorthand notation for code which resembles traditional scientific code is also possible by using the same instance of a variable for accessing its data in cells:

```
const Mass_Density Rho{};
const Background_Magnetic_Field B0{};
cell_data[Rho] = ...;
cell_data[B0][0] = ...;
cell_data[B0][1] = ...;
...
```

The possibility of using a generic simulation cell approach in the traditional high-performance language of choice – Fortran – seems unlikely as Fortran 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.

8 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 and parallel performance is negligible.

Acknowledgements. Alex Glocer for insightful discussions and the NASA Postdoctoral Program for financial support.

References

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: 15 July 2014), 2012. 4581

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. 4581

Hill, C., DeLuca, C., Balaji, V., Suarez, M., and Silva, A. D.: The architecture of the earth system modeling framework, Comp. Sci. Eng., 6, 18–28, doi:10.1109/MCISE.2004.1255817, 2004. 4580

Honkonen, I., von Alfthan, S., Sandroos, A., Janhunen, P., and Palmroth, M.: Parallel grid library for rapid and flexible simulation development, Comp. Phys. Commun., 184, 1297–1309, doi:10.1016/j.cpc.2012.12.017, 2013. 4583, 4584, 4585, 4590, 4599, 4602

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. 4580

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. 4580

McCormack, D.: Generic programming in Fortran with Forpedo, SIGPLAN Fortran Forum, 24, 18–29, doi:10.1145/1080399.1080401, 2005. 4592

Miller, G.: A scientist's nightmare: software problem leads to five retractions, Science, 314, 1856–1857, doi:10.1126/science.314.5807.1856, 2006. 4579

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, Springer, Berlin, Heidelberg, doi:10.1007/3-540-51084-2_2, 13–25, available at: http://www.stepanovpapers.com/genprog.ps, 1989. 4579

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. 4579

Post, D. E. and Votta, L. G.: Computational science demands a new paradigm, Phys. Today, 58, 35–41, doi:10.1063/1.1881898, 2005. 4578

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. 4580

Stroustrup, B.: Learning standard C++ as a new language, C/C++ Users J., 17, 43–54, available at: http://dl.acm.org/citation.cfm?id=315554.315565, 1999. 4579

Thomas, W. M., Delis, A., and Basili, V. R.: An analysis of errors in a reuse-oriented development environment, J. Syst. Software, 38, 211–224, doi:10.1016/S0164-1212(96)00152-5, 1997. 4579

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. 4580

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. 4579

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, ACM, New York, NY, USA, 9:1–9:7, doi:10.1145/2349896.2349905, 2012. 4591

Zaghi, S.: OFF, Open source Finite volume Fluid dynamics code: a free, high-order solver based on parallel, modular, object-oriented Fortran {API}, Comput. Phys. Commun., 185, 2151–2194, doi:10.1016/j.cpc.2014.04.005, 2014. 4592

Zhang, L. and Parashar, M.: Seine: a dynamic geometry-based shared space interaction framework for parallel scientific applications, in: Proceedings of High Performance Computing – HiPC 2004: 11th International Conference, Springer LNCS, 189–199, 2006. 4580

Table 1. Run times of serial GoL programs using different cell types and memory layouts for their variables compiled with GCC 4.8.2.

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

```
template <class ... Variables> class Cell;

template <class Variable> class Cell<Variable> {
private:
    typename Variable::data_type data;

public:
    typename Variable::data_type& operator[](
        const Variable&
    ) {
        return this->data;
    }
};

template <
    class Current_Variable,
    class ... Rest_Of_Variables
> class Cell<
    Current_Variable,
    Rest_Of_Variables...
> : public Cell<Rest_Of_Variables...> {
private:
    typename Current_Variable::data_type data;

public:
    using Cell<Rest_Of_Variables...>::operator[];

    typename Current_Variable::data_type& operator[](
        const Current_Variable&
    ) {
        return this->data;
    }
};
```

Figure 1. Serial implementation of the generic simulation cell class that is not const-correct but is otherwise complete.

```cpp
#include "array"
#include "cstdlib"
#include "gensimcell.hpp"

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

int main() {
    constexpr size_t 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_Neighbors()] = 0;
        if (rand() < RAND_MAX / 10)
            cell[Is_Alive()] = true;
        else
            cell[Is_Alive()] = false;
    }}

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

            // neighbor index offsets: +1, 0, -1
            for (auto row_offset: {1ul, 0ul, width - 1}) {
            for (auto col_offset: {1ul, 0ul, height - 1}) {
                if (row_offset == 0 and col_offset == 0)
                    continue;
                // periodic boundaries
                const auto& neighbor
                    = grid[
                        (row + row_offset) % height
                    ][
                        (col + col_offset) % width
                    ];

                if (neighbor[Is_Alive()])
                    cell[Live_Neighbors()]++;
            }}
        }}
        // set new state
        for (size_t row = 0; row < height; row++) {
        for (size_t col = 0; col < width; col++) {
            Cell_T& cell = grid[row][col];

            if (cell[Live_Neighbors()] == 3)
                cell[Is_Alive()] = true;
            else if (cell[Live_Neighbors()] != 2)
                cell[Is_Alive()] = false;

            cell[Live_Neighbors()] = 0;
        }}
    }
    return 0;
}
```

Figure 2. A serial program playing Conway's Game of Life implemented using the generic simulation cell class.

```cpp
/* advection */
struct Density { using data_type = double; };
struct Density_Flux { using data_type = double; };
struct Velocity { using data_type = std::array<double, 2>; };
using Cell_T = gensimcell::Cell<Density, Density_Flux, Velocity>;
using Grid_T = array<array<Cell_T, width>, height>;

void apply_solution(Grid_T& grid) {
    for (auto& row: grid) {
    for (auto& cell: row) {
        cell[Density()] += cell[Density_Flux()];
        cell[Density_Flux()] = 0;
    }}
}

/* particle propagation */
struct Velocity { using data_type = array<double, 2>; };
struct Particles { using data_type = vector<array<double, 2>>; };
using Cell_T = gensimcell::Cell<Velocity, Particles> Cell_T;
using Grid_T = array<array<Cell_T, width>, height>;

void initialize(Grid_T& grid) {
    for (size_t row_i = 0; row_i < height; row_i++) {
    for (size_t cell_i = 0; cell_i < width; cell_i++) {

        const auto
            cell_center = get_cell_center(grid, {cell_i, row_i}),
            cell_size = get_cell_size(grid, {cell_i, row_i});

        auto& cell = grid[row_i][cell_i];

        cell[Particles()].push_back({
            cell_center[0] - cell_size[0] / 4,
            cell_center[1] - cell_size[1] / 4
        });
    }}
}
```

Figure 3. Excerpts from separate serial programs using the generic simulation cell class modeling advection and particle propagation in a prescribed velocity field.

```
...
#include "dccrg.hpp"
#include "dccrg_cartesian_geometry.hpp"
#include "gensimcell.hpp"
...
int main(int argc, char* argv[]) {
    using Cell = gol::Cell;
    if (MPI_Init(&argc, &argv) != MPI_SUCCESS) {...}

    dccrg::Dccrg<Cell, dccrg::Cartesian_Geometry> grid;
    ...
    gol::initialize<Cell, gol::Is_Alive, gol::Live_Neighbors>(grid);

    const std::vector<uint64_t>
        inner_cells = grid.get_local_cells_not_on_process_boundary(),
        outer_cells = grid.get_local_cells_on_process_boundary();
    ...
    while (simulation_time <= M_PI) {
        ...
        Cell::set_transfer_all(true, gol::Is_Alive());
        grid.start_remote_neighbor_copy_updates();
        gol::solve<
            Cell,
            gol::Is_Alive,
            gol::Live_Neighbors
        >(inner_cells, grid);
        grid.wait_remote_neighbor_copy_update_receives();
        gol::solve<...>(outer_cells, grid);
        gol::apply_solution<...>(inner_cells, grid);
        grid.wait_remote_neighbor_copy_update_sends();
        gol::apply_solution<...>(outer_cells, grid);
        simulation_time += time_step;
    }
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

Figure 4. Excerpts from a parallel program playing Conway's Game of Life implemented using the generic simulation cell class and the dccrg grid library (Honkonen et al., 2013).

```
struct Internal_Particle_Storage {
    std::vector<std::array<double, 3>> coordinates;

    std::tuple<
        void*, int, MPI_Datatype
    > get_mpi_datatype() const {
        return std::make_tuple(
            (void*) this->coordinates.data(),
            3 * this->coordinates.size(),
            MPI_DOUBLE
        );
    }
    void resize(const size_t new_size) {
        this->coordinates.resize(new_size);
    }
};

struct Internal_Particles {
    using data_type = Internal_Particle_Storage;
};

struct External_Particle_Storage {
    std::vector<std::array<double, 3>> coordinates;
    std::vector<uint64_t> destinations;

    std::tuple<
        void*, int, MPI_Datatype
    > get_mpi_datatype() const {
        std::array<int, 2> counts{
            3 * this->coordinates.size(),
            1 * this->destinations.size()
        }
        std::array<MPI_Aint, 2> displacements{
            0, reinterpret_cast<
                const char* const
            >(this->destinations.data())
            - reinterpret_cast<
                const char* const
            >(this->coordinates.data())
        };
        std::array<MPI_Datatype, 2> datatypes{
            MPI_DOUBLE, MPI_UINT64_T
        }

        MPI_Datatype final_datatype;
        MPI_Type_create_struct(
            2, counts.data(), displacements.data(),
            datatypes.data(), &final_datatype
        );

        return std::make_tuple(
            (void*) this->coordinates.data(),
            1, final_datatype;
        );
    }

    void resize(const size_t new_size) {
        this->coordinates.resize(new_size);
        this->destinations.resize(new_size);
    }
};

struct External_Particles {
    using data_type = External_Particle_Storage;
};
```

Figure 5. Variables used in the example parallel particle propagation model showing the logic of providing MPI transfer information for the variables' data.

```
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 }
```

Figure 6. 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).

```
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 {
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 };
```

Figure 7. An example of converting existing software to use an application programming interface (API) identical to the generic cell class. The cell class defined on lines 1..15 is usable as is e.g. with dccrg (Honkonen et al., 2013). API conversion consists of adding empty classes (lines 17 and 18) for denoting simulation variables, and adding [] operators (lines 22..28) for accessing the variables' data.