Computer Physics Communications 226 (2018) 151–164

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Schnek: A C++ library for the development of parallel simulation codes on regular grids☆

Holger Schmitz

Central Laser Facility, STFC, Rutherford Appleton Laboratory, Didcot, Oxon., OX11 0QX, United Kingdom

ARTICLE INFO

Article history:
Received 15 May 2017
Received in revised form 8 November 2017
Accepted 27 December 2017
Available online 2 February 2018

Keywords:
Numerical simulation
Library
Parallel
Regular grid

ABSTRACT

A large number of algorithms across the field of computational physics are formulated on grids with a regular topology. We present Schnek, a library that enables fast development of parallel simulations on regular grids. Schnek contains a number of easy-to-use modules that greatly reduce the amount of administrative code for large-scale simulation codes. The library provides an interface for reading simulation setup files with a hierarchical structure. The structure of the setup file is translated into a hierarchy of simulation modules that the developer can specify. The reader parses and evaluates mathematical expressions and initialises variables or grid data. This enables developers to write modular and flexible simulation codes with minimal effort. Regular grids of arbitrary dimension are defined as well as mechanisms for defining physical domain sizes, grid staggering, and ghost cells on these grids. Ghost cells can be exchanged between neighbouring processes using MPI with a simple interface. The grid data can easily be written into HDF5 files using serial or parallel I/O.

Program summary
Program Title: Schnek
Program Files doi: http://dx.doi.org/10.17632/ps6cmgdg8p.1
Licensing provisions: GNU GPLv3
Programming language: C++
External routines/libraries: Boost, MPI (optional), HDF5 (optional)
Nature of problem: Flexible large-scale simulation codes require a large amount of administrative code. This code is tedious to implement and often requires knowledge about software design that falls outside of the expertise of the implementing scientist.
Solution method: Implementation of a powerful but easy-to-use library that provides much of the administrative tasks and creates a template for the development of simulation codes. MPI is used for parallelisation of data on regular grids. Simulation parameters can be read from setup files using a powerful syntax.
Restrictions: Optimised for regular grids

1. Introduction

In recent years the use of highly parallel computers for the simulation of physical phenomena has greatly increased. While the fastest supercomputer has exceeded 10 million cores [1], many universities and research facilities run their own distributed-memory computer clusters. This development produces a need for parallel simulation codes that scale well from just a few cores up to the largest computers. There is a large variety of numerical schemes used to simulate different physical systems. Many different large scale simulations codes face similar problems concerning administrative tasks, such as reading setup data, managing data transfer between processes in a parallel execution environment, or saving the results of the simulation into standardised output files. For any new simulation code a decision has to be made, how much effort is spent writing code for these administrative tasks.

In one extreme this results in highly specialised codes where little thought is given to making it useable or flexible. Adapting or extending the code for new tasks is difficult and time consuming and usage of the resulting software often remains limited to a small

☆ This paper and its associated computer programme are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

E-mail address: holger.schmitz@stfc.ac.uk.

https://doi.org/10.1016/j.cpc.2017.12.023
0010-4655/

group around the original developer. If, on the other hand, a more flexible and reusable code is desired, much effort has to be spent on writing administrative code. Given that many codes are written as part of research projects, where the main goal is to produce scientific output, there is often little incentive to spend the extra effort. In addition it is very likely that the scientist writing the code has no solid background in software development techniques. The programming skills of most computational scientists are self taught, often leading to poor software design [2]. This dilemma can be solved with code libraries that implement the administrative tasks and provide a framework for the simulation code. In order to be useful such a library has to be both flexible and easy-to-use. This implies that a programming language should be used that combines computational speed with flexibility and expressive power. Over the last two decades C++ [3] has been established as one of the main languages for numerical simulation codes. It supports object oriented and generic programming concepts while maintaining high performance [4].

In the past there have been a number of C++ libraries that address the problem of providing administrative tasks or avoiding repetitive boiler-plate code. The `glsim` library [5] stems from an attempt to clearly define the administrative tasks that are needed for any numerical simulation software. The result is a very generic library addressing these tasks. However, because of the fact that only few assumptions were made the library covers only a small part of the tasks that are needed by a realistic simulation code. On the other end of the spectrum there are a large number of libraries that focus on small but well-defined subsets of the tasks. Many of these libraries are concerned with representing mathematical concepts such as Blitz++ [6] or the Matrix Template Library [7]. Others focus on specific techniques used in numerical simulations. The adaptive mesh refinement method is one example of such a technique and libraries that attempt to address adaptive mesh refinement in general are, among many others, the DCCRG library [8] and the STAPL library [9]. These libraries usually remain isolated attempts to solve a small and well-defined problem. They do not, however, provide a unified framework for writing complete simulation codes. Schnek is an attempt to provide such a framework for simulations based on regular grids. Schnek is a simple and easy-to-use library that addresses a variety of different problems faced by the developer of a simulation code. The library provides classes for representing multi-dimensional regular grids and distributing them over multiple processes using the MPI Message Passing Interface [10]. The grids can easily be saved to HDF5 data files which provide a cross platform standard for storing grid data [11]. A major feature of Schnek is that the simulation can be configured through a setup file that not only allows the end user to specify values for simulation parameters but also provides a method of controlling modules of the simulation code in a hierarchical layout. Here and in the rest of this article, *developer* means the person developing a simulations code on top of Schnek or other existing libraries. The term *user* designates the end user of the simulation code. In many situations the user is different from the developer and may have little knowledge in code development. In order to encourage collaborative code projects, modules can provide literature references.

There are a number of other libraries available that at first sight seem to provide similar functionality. Most notably, PETSc [12] is a library providing a wealth of routines and algorithms for solving linear and non-linear matrix equations. PETSc provides routines for parallelisation and even a mechanism to produce literature references. Libraries, such as Trilinos [13] or Hypre [14] also provide a range of Matrix solvers but little or no support for handling multidimensional computational grids representing physical domains. Other projects, like Kokkos [15], aim at providing a mechanism to create parallelised code, automatically optimised for different architectures. The resulting code can be run effectively on computer clusters, multi core CPUs, and on GPU architectures. Schnek differs from PETSc and other libraries in a number of ways. Specifically, Schnek provides classes for creating *modular* simulation codes. Modules can contain anything from physical effects or boundary conditions, through variations of different solvers, to output or post-processing of data. To illustrate this with an example, consider a Particle-In-Cell type code [16]. Here modules could contain different particle collision models, variations of the particle push algorithm, boundary conditions to inject or reflect particles, or the averaging of the particle distribution function for diagnostics. Schnek then provides a parser for translating setup files into a hierarchy of modules at run time. To this end Schnek defines a Domain Specific Language (DSL) with a C-style syntax through which the developer can expose modules and parameters to the end-user. Module data, including arbitrary dimensional grids, can be initialised from a setup file that can include mathematical expressions involving the physical coordinates or the simulation time. Schnek's literature database is tied in with the modular simulation structure, ensuring only references relevant to a specific run are written out. Schnek, therefore, provides functionality not provided by previous libraries allowing a developer to make use of a general DSL for their specific simulation code. In this way Schnek solves a different problem and can complement libraries such as PETSc, Trilinos, or Hypre.

In addition to its dependence on the MPI and HDF5 libraries Schnek makes use of the Boost C++ library [17]. This does not, however, pose a serious restriction as Boost is available on most systems and has become a de-facto standard for C++ development. Schnek restricts itself to the use of the most established parts of the Boost library in order to avoid version incompatibilities. Using the Schnek library, a single developer can quickly develop a flexible parallel simulation code which can easily be adapted to changing problem domains, can be shared among researchers within a group, or can be opened up to the wider community. The applicability of Schnek has been demonstrated by a variety of full scale simulation codes as well as a number of proof of principle codes. These codes include a fluid code, a non-linear full Maxwell solver, and a Particle-in-Cell code for plasma simulations [18]. Schnek is available on GitHub at https://github.com/holgerschmitz/Schnek.

This paper is structured as follows. Section 2 describes the `Grid`, `Field` and `Array` classes that represent multi-dimensional grids and fixed sized arrays. Section 3 outlines the approach to parallelisation in Schnek. Using MPI, values in the ghost cells of a grid can easily be updated by the data from neighbouring processes. Section 4 describes the processing of simulation setup files. Setup files allow the end user to define simulation parameters using mathematical expressions. The declaration of user variables inside the setup file is supported. The hierarchical structure of the file is translated into a hierarchical structure of simulation modules, so called blocks. Section 5 provides an overview of the literature system to encourage collaborative code development in the scientific community. The examples provided throughout Sections 2–5 make up the skeleton of a simulation code that uses the Finite Difference Time Domain (FDTD) method to solve Maxwell's equations in a medium with fixed relative permittivity on a Yee grid [19]. Since Schnek is a general purpose library that does not contain any specific numerical algorithms, it is not possible to benchmark the code directly. Instead, Section 6 presents benchmarks and scalability runs of the FDTD code developed in the preceding sections. A summary and conclusions are given in Section 7.

2. Grid and array class templates

2.1. Multidimensional grids

The C++ language provides little support for multi-dimensional arrays and the representation of multi-dimensional regular grids. Schnek provides this support in form of the `Grid` class template. The class takes two to four template arguments. The first two arguments specify the data type contained in the grid and the rank of the grid. The third and fourth template arguments are optional and specify the range checking policy and the data storage model. The `Grid` class defines an internal type called `IndexType` which is used to specify indices on the grid and access data elements. It is possible to create grids with arbitrary index ranges by specifying a lower and an upper range index.

Listing 1: Constructing two `Grid` objects

```
typedef Grid<double, 3>::IndexType IndexType;
Grid<double, 3> grid(IndexType(5, 7, 10));
Grid<double, 3> gridB(IndexType(2, 2, 2), IndexType(10,10,10));
```

Listing 1 shows how grids can be created using the two different constructors. Both grids have a rank of 3 and contain `double` floating point values. The first grid is constructed by specifying the three components of the size of the grid. The valid index range starts at (0, 0, 0) and ends at, including (4, 6, 9), thus adopting the C convention of indexing arrays. The second grid is constructed by specifying the components of the lower and the upper bound of the valid index range directly. After a grid has been constructed it can be resized by calling the `resize` method.

Access to the elements in the `Grid` is given through the function operator, e.g. `grid(i,j,k)`. The function operator takes the components of the index position as arguments and can, depending on the context, return an rvalue, i.e. a value that can be used in an expression, or an lvalue, i.e. a reference that a value can be assigned to. The interface of the `Grid` class defines the function operator for grids up to rank 10. For comparison, Fortran 90 defines a maximum rank of 7 while for Fortran 2008 this has increased to 15. For ranks larger than 10 the index operator allows access to the elements, e.g. `grid[pos]`.

Listing 2: Accessing elements of a `Grid` object

```
grid = 1.0;
grid(2, 3, 7) = 2.5;
IndexType pos(3, 4, 5);
grid[pos] *= 2.0;  // access through the index operator
```

In listing 2 the different methods of accessing the elements in a grid are shown. The first line initialises all elements of the grid with a value of 1.0. The second line assigns a value of 2.5 to the grid element at position (2, 3, 7), while the third and the fourth line multiply the grid element at position (3, 4, 5) by 2. Listing 3 shows two examples of typical loops over all the elements in the grid advancing Ampere's law and Faraday's law in a medium with relative permittivity `eps_rel` using the FDTD method. The member functions `getLo` and `getHi` return the lower and upper limits of the valid index range. In `stepE` the iteration is written as a triple nested loop. In `stepB` the same result is achieved by constructing a `Range` over the index domain and using an iterator to traverse all elements inside the range. The iterator can be used in conjunction with the index operator to access the grid elements.

Listing 3: FDTD formulation of Ampere's law in a medium with relative permittivity `eps_rel`

```
void stepE(double dt) {
  IndexType low = Ex.getLo();
  IndexType high = Ex.getHi();

  for (int i=low[0]+1; i<=high[0]; ++i)
    for (int j=low[1]+1; j<=high[1]; ++j)
      for (int k=low[2]+1; k<=high[2]; ++k) {
        Ex(i,j,k) = Ex(i,j,k)
            + dt*clight2/eps_rel*( (Bz(i,j,k) - Bz(i,j-1,k))/dx[1]
                                 - (By(i,j,k) - By(i,j,k-1))/dx[2] );
        Ey(i,j,k) = Ey(i,j,k)
            + dt*clight2/eps_rel*( (Bx(i,j,k) - Bx(i,j,k-1))/dx[2]
                                 - (Bz(i,j,k) - Bz(i-1,j,k))/dx[0] );
        Ez(i,j,k) = Ez(i,j,k)
            + dt*clight2/eps_rel*( (By(i,j,k) - By(i-1,j,k))/dx[0]
                                 - (Bx(i,j,k) - Bx(i,j-1,k))/dx[1] );
  }
}

void stepB(double dt) {
  Range<int,3> range(Bx.getLo(), Bx.getHi());
  IndexType unit_x(1,0,0), unit_y(0,1,0), unit_z(0,0,1);

  for (Range<int,3>::iterator p=range.begin(); p!=range.end(); ++p) {
    Bx[*p] += dt*( (Ey[*p + unit_z] - Ey[*p])/dx[2] - (Ez[*p + unit_y] - Ez[*p])/dx[1] );
    By[*p] += dt*( (Ez[*p + unit_x] - Ez[*p])/dx[0] - (Ex[*p + unit_z] - Ex[*p])/dx[2] );
    Bz[*p] += dt*( (Ex[*p + unit_y] - Ex[*p])/dx[1] - (Ey[*p + unit_x] - Ey[*p])/dx[0] );
  }
}
```

By default the access functions of the `Grid` class do not check the value of the index against the valid index range. This default has been chosen for sake of speed of the resulting code. However, accessing elements outside of the valid range can lead to errors which are hard to find. During development of simulation codes it is often advantageous to check the indices against the valid range. The third template argument of the `Grid` class allows the specification of an index checking policy. The checking policy is a class that defines a static `check` method. This method is called every time an element of the grid is accessed. The `check` method of the default `GridNoArgCheck` class does not perform any test on the arguments. By defining the method `inline`, the compiler can optimise the function call away. An alternative to `GridNoArgCheck` is the `GridAssertCheck` class. The `check` method of this class produces a run-time assertion failure if the index is out of range. It is good practice to declare typedefs in the application header defining the `Grid` used throughout the simulation. A debug macro can then be used to switch between a grid with range checking and a grid without range checking.

Listing 4: Usage of `GridAssertCheck` to check the out-of range indices.

```
Grid<double, 3, GridAssertCheck> gridCheck(IndexType(3, 3, 3));

// causes runtime assertion failure
gridCheck(2,2,5) = 1.0;
```

In some cases fine control over the internal memory layout of the grid is needed. By default the `Grid` class will use a standard C style memory layout. Multi-dimensional C arrays use a Row-Major Order which means that an increment in the last component of the index position corresponds to an increment in the memory address. Fortran arrays, on the other hand use Column-Major Order, i.e. the first component of the index position corresponds to an increment in the memory address. The fourth template argument of the `Grid` class is used to specify the storage model. The template argument `SingleArrayGridStorageFortran` is used to specify Fortran ordering, as can be seen in listing 5.

Listing 5: Specifying Fortran memory layout

```
Grid<double, 3,
     GridNoArgCheck,
     SingleArrayGridStorageFortran> gridF(IndexType(3, 3, 3));

double *data = gridF.getRawData();
```

The listing also shows that access to the underlying raw data is granted through the `getRawData` method. This method breaks the encapsulation of the `Grid` class by exposing implementation details to the user of the class. The choice has been made to break encapsulation because it might sometimes be necessary to pass pointers to the raw data to third party libraries, such as Hypre [14] or FFTW [20]. The alternative would be to require the user to copy the data into a separate low level array. This can incur a large memory overhead and speed penalty which is not acceptable for many applications.

In addition to the `SingleArrayGridStorageFortran` memory model, Schnek also defines a lazy allocation memory model through the `SingleArrayLazyAllocation` class. This model reduces the penalty for frequent resizing of the grid by allocating larger chunks of memory when the grid size grows and by delaying the deallocation of the memory when the grid size decreases. A usage example for this storage model might be the exchange of particle data between processes in a particle-in-cell code. A `Grid` with rank 1 can be used as a buffer to store and exchange particle data, see Section 3. The number of particles that need to be passed to a neighbouring process can fluctuate during the course of the simulation, requiring the buffer to be resized at every time step. The `SingleArrayLazyAllocation` memory model reduces the number of underlying memory deallocation and allocation requests by internally allocating a buffer slightly larger than required.

2.2. Fixed length arrays

In the previous section we have used the `IndexType`, defined within the `Grid` class, to create grids and access their elements. The `IndexType` is a convenience `typedef` to the `Array` class template. This class embodies the concept of a fixed length, one dimensional array. The class takes up to three template arguments. The first two template arguments specify the type and size of the array. An optional third template argument specifies the range checking policy, similar to the range checking policy in the `Grid` class.

Listing 6: Usage of the `Array` class template

```
Array<double, 3> a;
Array<double, 3> b(1.0, 2.0, 3.0);
Array<std::string, 5> s("a","b","c","d","e");

for (size_t i = 0; i < 3; ++i)  a[i] = 2 * i + 1;

Array<double, 3> c;
c = 2.0*a + b;
```

Listing 6 shows typical usage of the `Array` class. The constructor can take n arguments, where n is the size of the array. These values are used to initialise the content of the array. The index operator allows access to the elements of the array. The last line in listing 6 shows how the `Array` class can be used in mathematical vector expressions. Schnek uses expression templates [21] to avoid unnecessary creation of temporary objects and improve performance of `Array` expressions.

2.3. Stagger, ghost cells and fields

The `Grid` class provides basic storage for multi-dimensional grid data. In numerical simulations of physical systems the grid corresponds to a domain, either in space or momentum, or in some other physical variable. In order to initialise the simulation grid or to analyse the data it is necessary to map the integer grid coordinates to the coordinates in the physical domain. The problem of mapping between the coordinate spaces is complicated by the fact that some simulation techniques require the numerical grids to be staggered. This means that physical location of the numerical points of one grid is offset against the location of the other grid by a fraction of a grid cell. An example is the Yee scheme for the FDTD method which approximates Maxwell's equations on a numerical grid [19]. Here the grids of the components of the electric field are staggered in the direction of the respective component, i.e. the E_x grid is staggered in the x-direction and so on. The grids of the components of the magnetic field, on the other hand, are staggered in the two directions transverse to the respective component, i.e. the B_x grid is staggered in the y- and z-directions.

In addition to grid staggering, the implementation of boundary conditions often requires the existence of so called *ghost cells*. Numerical schemes generally update values of a grid cell by using the values of the neighbouring grid cells. On the boundaries of the simulation domain this update requires grid cells that lie outside of the simulation domain. These ghost cells are updated according to a separate algorithm and do not take part in the update of the inner simulation domain. The ghost cells are part of the numerical grid but lie outside of the physical simulation domain. This adds another layer of complexity to the mapping from the numerical grid coordinates to the physical coordinates. Although the mapping between these coordinates is linear, it can be different for different grids and it is the experience of the author that an incorrect implementation of these transformations can lead to errors which are extremely hard to find.

The `Field` class extends the `Grid` class and encapsulates the concepts of a physical domain size, grid staggering and ghost cells. A `Field` is initialised with the grid size of the simulation domain and a `Range` representing the physical domain. The grid stagger is represented by an `Array` of Boolean values. Currently only staggering by half a grid cell is supported. Listing 7 shows how `Field` objects for simulating the electromagnetic fields on a Yee grid can be initialised. The physical domain is given by $-5\text{ mm} \leq x, y, z \leq 5\text{ mm}$ and the thickness of the ghost cell border is 2. After initialisation, the `positionToIndex` and `indexToPosition` methods can be used to convert between physical coordinates and grid coordinates. The physical size, the grid staggering, and the number of ghost cells influence the numerical interpretation of the values stored in the grid. They do not change the internal layout. For this reason it was decided to pass these parameters to the constructor, rather than specifying them as template parameters.

Listing 7: Creating `Field` objects for simulating the electromagnetic fields on a Yee grid.

```
int ghostCells = 2;

IndexType lo(0,0,0), hi(100,100,100);
Array<double, 3> physLo(-5e-3, -5e-3, -5e-3), physHi(5e-3, 5e-3, 5e-3);
Range<double, 3> physRange(physLo, physHi);

Array<bool, 3> exStaggerYee(true,  false, false);
Array<bool, 3> eyStaggerYee(false, true,  false);
Array<bool, 3> ezStaggerYee(false, false, true );

Array<bool, 3> bxStaggerYee(false, true,  true );
Array<bool, 3> byStaggerYee(true,  false, true );
Array<bool, 3> bzStaggerYee(true,  true,  false);

Field<double, 3> Ex(lo, hi, physRange, exStaggerYee, ghostCells);
Field<double, 3> Ey(lo, hi, physRange, eyStaggerYee, ghostCells);
Field<double, 3> Ez(lo, hi, physRange, ezStaggerYee, ghostCells);

Field<double, 3> Bx(lo, hi, physRange, bxStaggerYee, ghostCells);
Field<double, 3> By(lo, hi, physRange, byStaggerYee, ghostCells);
Field<double, 3> Bz(lo, hi, physRange, bzStaggerYee, ghostCells);
```

3. Parallelisation using MPI

Large-scale simulation codes need to make use of modern parallel computer architectures. One of the main approaches of writing parallel codes is the use of the Message Passing Interface (MPI) library. Schnek provides a mechanism for distributing grids onto multiple processes using the MPI library. At the core of this mechanism is the `MPICartSubdivision` class. After initialising MPI with a call to `MPI_Init` the `MPICartSubdivision` class can be initialised with the size of the global grid and the thickness of the ghost cell layer. `MPICartSubdivision` will attempt to create an optimal partitioning of the global simulation domain on to the available processes.

If the global simulation domain is a D-dimensional hypercube, i.e. the size is identical in all D dimensions, this is achieved by finding D integer factors of the total number N of processes. The factors are chosen to be as close to $N^{1/D}$ as possible. This ensures that the local simulation domain on a single process is close to a cube shape and thus optimises the volume to surface ratio. Because the grids exchange data on the surface of the local domains this optimisation reduces the communication between the processes. In the case where the grid size is different in the different dimensions the algorithm will scale the factors so as to keep close to a cube shape on each processor. After initialisation the global grid domain has been split and each process will operate on a rectangular sub-domain. `MPICartSubdivision` provides a number of accessors giving information about the local domain. The methods `getLo` and `getHi` return the extent of the local grid, including the ghost cells. These can be used to initialise a `Grid` object. To initialise a `Field` object the inner extent of the local grid is needed. This can be obtained from the `getInnerLo` and `getInnerHi` member functions of `MPICartSubdivision`. Another useful function for initialising or resizing `Field`s is the `getInnerExtent` member function of `MPICartSubdivision`. This function takes the global physical domain size as input and returns the physical range of the local subdomain. Listing 8 shows how `MPICartSubdivision` can be used to set up local fields.

Listing 8: Splitting the global simulation domain on to the individual processes using `MPICartSubdivision`.

```
int ghostCells = 2;
Array<int, 3> gridSize(1000, 1000, 1000);
Array<double, 3> domainSize(1.0, 1.0, 1.0)

MPICartSubdivision<Field> subdivision;

subdivision.init(gridSize, ghostCells);

Array<int, 3> lo  = subdivision.getInnerLo();
Array<int, 3> hi = subdivision.getInnerHi();

Range<double, 3> physRange = subdivision.getInnerExtent(domainSize);

Ex.resize(lo, hi, physRange, exStaggerYee, ghostCells);
Ey.resize(lo, hi, physRange, eyStaggerYee, ghostCells);
Ez.resize(lo, hi, physRange, ezStaggerYee, ghostCells);

Bx.resize(lo, hi, physRange, bxStaggerYee, ghostCells);
By.resize(lo, hi, physRange, byStaggerYee, ghostCells);
Bz.resize(lo, hi, physRange, bzStaggerYee, ghostCells);
```

After initialisation and setting up of the fields the `MPICartSubdivision` class is responsible for the data communication between the MPI processes. The `exchange(Field &F, int d)` member function is used to exchange ghost cell data between neighbours in the dimension given by `d`. The grid cells in the inner region on one process are disjoint from the cells in the inner region of the neighbouring processes. On the other hand, the ghost cells of one process overlap with the inner regions of the neighbouring processes. This means that each grid cell in the ghost cell region has a corresponding inner cell in a neighbouring process. The `exchange(Field &F, int d)` member function of `MPICartSubdivision` copies values from the inner cells of the neighbours in the d-direction into the ghost cells of a field. Here d is between zero and the rank of the field. On the outer boundaries the values are circularly wrapped. Often the exchange needs to be carried out for all dimensions $d = 0, \ldots, \text{rank} - 1$. This can conveniently be done by calling `exchange(Field &F)` without the second argument. After a call to this function all ghost cells have been filled with the interior cells of the neighbouring processes. A typical usage in the context of the FDTD code is shown in listing 9. Each field is passed to the exchange method individually. Some optimisation could be achieved by grouping fields and exchanging their ghost cell data together. It should be noted that in many real world applications fields have to be exchanged at different times. Only limited grouping is possible in these cases. Existing simulation codes that perform an exchange for each grid individually have been shown to possess good scaling up to 100,000 s of cores [22,23].

Listing 9: Performing a simulation step and exchanging ghost cells. Here `subdivision` is assumed to be the `MPICartSubdivision` object created in listing 8 and `stepD` and `stepB` refer to listing 3.

```
void stepScheme(double dt) {
  stepD(dt);
  subdivision.exchange(Ex);
  subdivision.exchange(Ey);
  subdivision.exchange(Ez);
  stepB(dt);
  subdivision.exchange(Bx);
  subdivision.exchange(By);
  subdivision.exchange(Bz);
}
```

4. Setup files

One of the main tasks in writing flexible simulation codes is providing the ability to set up a variety of different configurations without the need to edit the source code or re-compile the software. A common approach is to read setup data from a file and set up the simulation according to the parameters specified in the file. In its simplest form the setup file can consist of a series of parameters in a fixed order. These parameters are read from a file and assigned to variables of the simulation code. Due to the lack of flexibility, this approach is clearly not adequate for medium or large-scale codes. A slightly improved approach creates name–value pairs for all parameters. The setup file then can specify a value for each name. While this allows parameters to be specified in any order it does not substantially improve maintainability. The reading of the parameters takes place in a single subroutine and any extension of the code that adds additional parameters will have to modify that routine.

Many simulation codes consist of a number of modules some of which are optional. The number of possible parameters to be specified can easily be in the hundreds. In this situation, a flat setup file is clearly not the optimal choice. It is also often desirable to specify one or more parameters as a function of another. This can make it much easier for the user to change configurations by simply changing one or two values in the setup file. Two main approaches have developed over time to solve this problem. One solution is to expose some part of the simulation code through an API. This API can be in a different language as the main code and Python is a common choice. The downside of this approach is that the devloper gives up control over the execution of the code to the end user. While being very flexible, it often requires programming experience of the end user. The other solution is the development of a Domain Specific Language (DSL). This is often more appropriate for simulation codes as it demands less programming knowledge from the end user. There are a large number of simulation codes defining their own DSL, ranging from simple grouped parameter definitions (e.g. EPOCH [22]) to complex languages (e.g. FreeFem++ [24]). In all cases, the DSL is specific to the simulation code and the developers of the respective codes had the task to implement the corresponding parsers. Schnek, on the other hand, provides routines and classes to any developer to incorporate Schnek's powerful DSL into their simulation code. Thus it relieves the developer of the task to develop their own DSL.

One of the most important features of Schnek is the ability to read hierarchically structured setup files that support mathematical expressions and translate these into an object hierarchy of simulation modules. The developer can expose modules and their parameters to the end user through Schnek's DSL. The parser that Schnek uses has been constructed using the Ragel tokenizer [25] and the Lemon parser generator [26]. This choice was made because both tools are lightweight and do not introduce additional library dependencies. A setup file is organised into hierarchical *blocks* represented by the `Block` class. At the top level, the setup file is represented by a root block. This root block, and each block that is contained within it, can contain setup data as well as an arbitrary number of additional blocks. Each block type can be associated with a C++ class that extends the `Block` class.

Listing 10: Reading a `SimulationBlock` from the setup file. The code shows a complete example including registering class attributes as setup parameters and reading the setup from a text file.

```
class SimulationBlock
   : public Block, public BlockContainer<FieldSolver> {
  private:
    double tMax;
    Array<int, 3> gridSize;
    Array<double, 3> size;
  protected:
    void initParameters(BlockParameters &parameters) {
      parameters.addConstant("pi", M_PI);
      parameters.addParameter("tMax", &tMax, 100.0);
      parameters.addArrayParameter("N", gridSize, 1000);
      parameters.addArrayParameter("L", size, 1.0);
    }
};

int main() {
  BlockClasses blocks;
  blocks("simulation").setClass<SimulationBlock>();

  std::ifstream in("example.setup");

  Parser P("example", "simulation", blocks);
  registerCMath(P.getFunctionRegistry());
  pBlock application = P.parse(in);

  SimulationBlock &simulation = dynamic_cast<SimulationBlock&>(*application);
  mysim.initAll();

  return 0;
}
```

Listing 10 shows a complete example that creates the specification for a flat setup file. The `SimulationBlock` class inherits from the `Block` class and defines a number of attributes. Note that `SimulationBlock` also inherits from `BlockContainer<FieldSolver>`. This inheritance will be explained in Section 4.1. The `initParameters` member function is a virtual function that will be called whenever an instance of `SimulationBlock` is created. This method can be overridden and used to register parameters. The `tMax` attribute is added by passing a string and the address of the variable. The string defines the name of the parameter in the setup file. In addition, `tMax` is given a default value of 100. This means that the specification of `tMax` can be omitted from the setup file in which case the default value will be used. It is also possible to register attributes of type `Array` with arbitrary rank as done with `gridSize` in the listing above. The string passed to the `addArrayParameter` function represents the base of the parameter name, in this case `"N"`. This will add three parameters with names `"Nx"`, `"Ny"`, and `"Nz"` corresponding to the three components of the array. By default the suffixes `"x"`, `"y"`, `"z"`, `"u"`, `"v"`, and `"w"` are appended to the base name for the 0th to 5th component of the `Array`. This behaviour can be changed by passing a string as optional fourth argument to `addArrayParameter`. The *n*th character in the string corresponds to the suffix of the *n*th component of the array. In addition to the initialisation of the array, the constant `pi` is added and initialised with the value of the `M_PI` macro.

In the `main` function the `SimulationBlock` class is registered and associated with the `"simulation"` name. the `BlockClasses` class is responsible for storing the associations between the name and the block class of a simulation block. The name of one of the registered blocks is then passed to the `Parser` defining the root class of the setup file. The `Parser` instantiates an object of the `SimulationBlock` class when reading the file and returns a smart pointer to the `Block` object. Schnek uses the convention of defining shared pointers to a class using the class name preceded by the letter `p`, such as `typedef boost::shared_ptr<Block>pBlock`. Because, by construction, the root block is of type `SimulationBlock`, the pointer can be downcast to the specialised type. Listing 11 shows a sample setup file that can be processed by the code from listing 10. The name of the `"simulation"` root block is not specified in the setup file as it is assumed implicitly that the content of the file lives within the root block. Note that it is possible to declare additional variables using a C style syntax, specifying the type and the variable name. These additional variables can be used in mathematical expressions.

Listing 11: Example of a setup file compatible with the code from listing 10

```
// comment starts with a double forward slash

// declaration and initialisation of local variables
float lambda = 1.05e-6;
float dx = lambda/20.0;

// definition of simulation parameters
Lx = 100e-6;
Ly = 100e-6;
```

```
Lz = 100e-6;

Nx = Lx/dx;
Ny = Ly/dy;
Nz = Lz/dz;

tMax = 2e-14;

// sin and cos function supplied through registerCMath
float vx = 0.5*clight*sin(pi/3);
float vy = 0.5*clight*cos(pi/3);
```

In listing 11 we observe that it is possible to use the `sin` and `cos` functions within the setup file. In fact, the call to `registerCMath` in listing 10 will register all pure functions present in the `<cmath>` header which is part of the C++ standard. However it is also possible to add custom functions. Listing 12 shows that this can be achieved in a single call to `registerFunction`. Listing 13 shows how the custom function can be used in the setup file.

Listing 12: Defining and adding a custom function to the function registry. The name `P` refers to the `Parser` object from listing 10

```
double normal(double x) {
  return exp(-0.5*x*x)/sqrt(2.0*M_PI);
}

[...]

P.getFunctionRegistry().registerFunction("normal",normal);
```

Listing 13: Using a custom function in the setup file.

```
float vx = normal(2.5);
```

4.1. Hierarchical structure

As was stated in the previous section, Schnek supports a hierarchical setup file format. Any block can contain an arbitrary number of child blocks. Listing 14 extends the example given in listing 10 by an additional `Block` class called `FieldSolver`. This class is designed to act as a child block of the `SimulationBlock` class. Here the term *child block* does not refer to inheritance but to a tree data structure in which a parent block contains one or multiple child blocks as data members. Note that `FieldSolver` does not inherit from the `Block` class directly, but instead from the `ChildBlock<FieldSolver>` class template. This class works together with the `BlockContainer<FieldSolver>` class introduced in listing 10. When instantiated, the `FieldSolver` will add itself to the `BlockContainer<FieldSolver>`. Via the `BlockContainer<FieldSolver>` inheritance the `SimulationBlock` gains a method called `childBlocks` through which it can access the `FieldSolver` members in a type safe way. The `BlockContainer<FieldSolver>` class is registered with the parser through the call to `blocks("FDTD"). setClass<FieldSolver>()`. This associates the block name `FDTD` with the `FieldSolver` class. The line `blocks("simulation").addChildren("FDTD")` registers the `FDTD` block as a child of the global simulation block `simulation`. This allows the user to use the `FDTD` name inside the global context in the setup file. Note that in this listing we also register a `FieldDiagnostic` class and add this as a child to the `simulation` block in the same way. This class will be discussed in Section 4.2.

Listing 14: Defining a hierarchical block structure. The remainder of the code is identical to listing 10

```
class FieldSolver : public ChildBlock<FieldSolver> {
  private:
    Field Ex, Ey, Ez;
    Field Bx, By, Bz;
    double eps_rel;

    void stepD(double dt);
    void stepB(double dt);
  protected:
    void initParameters(BlockParameters &blockPars) {
      parameters.addParameter("eps_rel", &eps_rel);
    }
  public:
    void stepScheme(double dt);
};

[...]

BlockClasses blocks;

blocks("simulation").setClass<SimulationBlock>();
blocks("FDTD").setClass<FieldSolver>();
blocks("Diagnostic").setClass<FieldDiagnostic>();

blocks("simulation").addChildren("FDTD")("Diagnostic");
```

Listing 15 shows how the hierarchical block structure is defined in the setup file. The example shows the definition of a `FDTD` block, defining the `eps_rel` attribute. The name of the block is specified after the block type. In the example the block has the name `solver`. The contents of the block are enclosed in curly brackets, resembling familiar C-style syntax. It is possible to define blocks to be nested to an arbitrary level.

Listing 15: Using the hierarchical block structure in the setup file.

```
FDTD solver {
  eps_rel = 1.2;
}
```

When this file is parsed, an instance of the `FieldSolver` class is created for every `FDTD` block and added as a child block to the global `SimulationBlock` object. The children of a block are accessible through a call to the `childBlocks` method and the parent of a block can be accessed through the `getParent` method. `childBlocks` returns an iterator range. The use of the `childBlocks` function can be seen in listing 18 which uses the `BOOST_FOREACH` macro supplied by the Boost library to iterate over all field solvers and call `stepScheme` on them. The hierarchical structure of the setup file results in a dynamic way of creating modular simulation codes. Each module is represented by a block and modules will only be instantiated if the user specifies their use in the setup file. In practice the child blocks can be virtual base classes, implemented by different inheriting classes. Each of these could implement a different algorithm for advancing the simulation, or they could implement different physical conditions or effects.

4.2. Diagnostic blocks

Schnek comes with a few pre-defined block classes that can be used for writing data. This includes a class `SimpleFileDiagnostic` which can be used to write data to text files. Any data object that defines the « operator for the `std::ostream` class can be written in this way. The `HDFGridDiagnostic` class provides a mechanism for writing data stored in `Grid` objects into HDF5 files. The class can be configured to use serial or parallel I/O with parallel I/O being the default. To use the `HDFGridDiagnostic` class template, one must create a simple class, inherititing from `HDFGridDiagnostic`. Only two virtual functions must be implemented, `getGlobalMin` and `getGlobalMax`. These need to return the global extent of the grid across all processes. In addition, all fields that should be available for diagnostic output need to be registered. This is done with a call to `addData` for each field. Note that this allows the diagnostic classes to access the fields but does not write them directly. Listing 16 shows the code needed to create an HDF diagnostic class and register the fields for output. The `FieldDiagnostic` can be added to the block hierarchy as shown in listing 14.

Listing 16: Adding an HDF5 diagnostic class and registration of fields to be used for diagnostics.

```
class FieldDiagnostic : public HDFGridDiagnostic<Field, Field*, DeltaTimeDiagnostic> {
  protected:
    Array<int, 3> getGlobalMin() { return Array<int, 3>(0); }
    Array<int, 3> getGlobalMax() { return globalMax; }
};

void FieldSolver::registerData() {
  addData("Ex", Ex);
  addData("Ey", Ey);
  addData("Ez", Ez);

  addData("Bx", Bx);
  addData("By", By);
  addData("Bz", Bz);
}
```

The last template parameter of the `HDFGridDiagnostic` class allows to choose between two different output models. `DeltaTimeDiagnostic` will create a diagnostic that produces output after a given physical time. `IntervalDiagnostic`, on the other hand, will write diagnostic output after a given number of time steps.

Listing 17: Adding diagnostic blocks to the setup file

```
float outDt = 1e-15;

Diagnostic Ey {
  field = "Ey";
  file = "Ey_#t.h5";
  deltaTime = outDt;
}

Diagnostic Bz {
  field = "Bz";
  file = "Bz_#t.h5";
  deltaTime = outDt;
}
```

The code in listing 16 allows a diagnostic block to be added to the setup file. This is shown in listing 17. Multiple diagnostic blocks can be added, each writing a different field. The `field` parameter specifies the data field that should be written. The string specified in the setup file must correspond to the string supplied to the `addData` function in listing 16. The `file` parameter specifies the name of the output file. The file name will be parsed and any appearance of `#t` will be replaced by a running index counting the output dumps. The `deltaTime` parameter specifies the simulation time interval after which output dumps should be generated.

In order to trigger the production of diagnostic output, the `DiagnosticManager` class must be used. This class is a singleton that is accessible through the static `instance` member function. Listing 18 demonstrates the usage of the `DiagnosticManager` in the main loop of the simulation.

Listing 18: The main loop of the simulation showing the use of the `DiagnosticManager`.

```
void SimulationBlock::execute() {
  time = 0.0;

  DiagnosticManager::instance().setPhysicalTime(&time);
  DiagnosticManager::instance().execute();

  double minDx = std::min(std::min(dx[0], dx[1]), dx[2]);
  dt = cflFactor*minDx/clight;

  while (time<=tMax) {
    if (subdivision.master())
      std::cout <<"Time␣"<< time << std::endl;

    BOOST_FOREACH(boost::shared_ptr<FieldSolver> f, childBlocks()) {
      f->stepScheme(dt);
    }

    time += dt;
    DiagnosticManager::instance().execute();
  }
}
```

4.3. Deferred evaluation of expressions

In some cases it is desirable to defer the evaluation of expressions from the setup file until some later time or to evaluate the expression multiple times. This is useful when some parameters of the expression are expected to change. One possible application is the definition of a time dependent parameter. For every time step in the simulation the expression should be re-evaluated with the updated time. Another common application is the initialisation of a grid with values that depend on the coordinates. It is possible in Schnek to declare variables that are read-only in the setup file which can be treated as independent variables. These can be used in expressions that do not evaluate to a constant but depend on the independent variables. An example is the use of the space coordinates x, y, and z, or the time t.

Listing 19: Reading fields from the setup file and filling them with data given by an expression that can depend on the coordinates x, y, and z. This piece of code extends the code shown in listing 14

```
class FieldSolver : public ChildBlock<FieldSolver> {
  private:
    Array<double, 3> x;
    Array<pParameter, 3> x_par, E_par, B_par;
    Array<double, 3> initE, initB;

  protected:
    initParameters(BlockParameters& parameters) {
      parameters.addParameter("eps_rel", &eps_rel);

      x_par = parameters.addArrayParameter("", x, BlockParameters::readonly);
      E_par = parameters.addArrayParameter("E", initE, 0.0);
      B_par = parameters.addArrayParameter("B", initB, 0.0);
    }

    init() {
      pBlockVariables blockVars = getVariables();
      pDependencyMap depMap(new DependencyMap(blockVars));
      DependencyUpdater updater(depMap);

      pParametersGroup spaceVars = pParametersGroup(new ParametersGroup());
      spaceVars->addArray(x_par);

      updater.addIndependentArray(x_par);

      fill_field(Ex, x, initE[0], updater, E_par[0]);
      fill_field(Ey, x, initE[1], updater, E_par[1]);
      fill_field(Ez, x, initE[2], updater, E_par[2]);

      fill_field(Bx, x, initB[0], updater, B_par[0]);
      fill_field(By, x, initB[1], updater, B_par[1]);
      fill_field(Bz, x, initB[2], updater, B_par[2]);
    }
    [...]
};
```

Listing 19 shows how these expressions can be used to fill the electromagnetic fields `Ex`, `Ey`, `Ez`, `Bx`, `By`, and `Bz` with values. The first step consists of keeping the `pParameter` objects that are returned by `addParameter` or `addArrayParameter` methods. Here `pParameter` is a Boost shared pointer to a `Parameter` object. It is not possible to add a `Grid` or `Field` directly as parameter. Instead, a single value parameter in case of `addParameter` or `Array` in case of `addArrayParameter` must be passed. In the case shown here `initE` and `initB` are used as intermediary variables. In a second step a `DepencyUpdater` is created that manages dependencies between variables in the setup file. Finally, the utility function `fill_field` initialises the data in the field from the expressions in the setup file. The function uses

the physical extent of the field and its stagger, which have been discussed in Section 2.3, to calculate the physical coordinates of the grid points. For each grid point the value of the components of the `x` array are set to the coordinates of the grid point. In the next step all variables from the setup file are evaluated that, directly or indirectly, depend on the `x` array and also, directly or indirectly, influence the value of the `initE` and `initB` attributes. Finally the values of `initE` and `initB` are updated and copied into the correct entry of the fields. Listing 20 shows a setup file that can be used to initialise the grid. Note that it is possible to define new variables that depend on x, y, and z and use these in the expression specifying the field values.

Listing 20: Setup file specifying the values on a `Field` using expressions involving the coordinates x and y

```
FDTD {
  eps_rel = 1.2;

  float amp = sin(2*pi*x/lambda);

  Ex = 0.0;
  Ey = amp;
  Ez = 0.0;

  Bx = 0.0;
  By = 0.0;
  Bz = sqrt(eps_rel)*amp/clight;
}
```

5. Literature system

The collaborative open source model has, in many fields of software development, been proven a successful approach for the creation of large-scale applications. From the projects such as the Linux kernel with over 1100 contributing developers [27] to small projects with only 2 or 3 contributors, collaborative software development has been responsible for some of the most successful software projects [28]. From the perspective of a scientist contributing to a project, the open source model has a major drawback. The contribution to a larger software package will, most likely, have resulted from a research project. The scientist will expect some form of recognition for the research that has been carried out in the development of the software. However, in large software projects with many contributions from different developers it might not be clear to the end user which algorithms have been used and who they were developed by. Small projects can solve this problem by providing documentation with a description of methods and algorithms together with literature references. For large modular codes this approach is not the ideal solution. The end user might have only used a few modules in a particular simulation run. Finding the information about the authors of these modules from a static documentation can be difficult and time consuming. Schnek provides a dynamic literature reference system. Each `Block` class, when initialised, can provide literature references and a short description of the module. When the simulation is run, the literature system can be used to write a BibTex file and a LaTeX file with references and descriptions of only those blocks that have been initialised through the setup file, i.e. only those block that are actually used in the simulation. Listing 21 shows how this is done in the context of the FDTD code. The code uses the Finite Difference Time Domain scheme for calculating the electromagnetic fields [19]. The `FieldSolver` that calculates these fields adds a reference to the corresponding article (top half of listing 21). In the `main` function (bottom half of listing 21) two files are written, `references.bib` contains the BibTex entries while `readme.tex` contains a document containing the descriptions.

Listing 21: Adding literature references to the `LiteratureManager` and writing information to BibTeX and LaTeXfiles.

```
void FieldSolver::init() {
  [...]

  LiteratureArticle Yee1966("Yee1966", "Yee, K",
      "Numerical solution of initial boundary value problems"
      "involving Maxwell's equations in isotropic media.",
      "IEEE Transactions on Antennas and Propagation", "1966", "AP-14", "302--307");

  LiteratureManager::instance().addReference(
      "Integration of electrodynamic fields uses the Finite Difference Time Domain method.",
      Yee1966);
}

[...]

std::ofstream referencesText("information.tex");
std::ofstream referencesBib("references.bib");

LiteratureManager::instance().writeInformation(referencesText,"references.bib");
LiteratureManager::instance().writeBibTex(referencesBib);
referencesText.close();
referencesBib.close();
```

6. Benchmarks and scalability

Schnek is a library that facilitates simulation codes on regular grids. As such it does not perform the major part of the numerical computations in a practical scenario. This means that it is not directly meaningful or even possible to benchmark Schnek on its own.

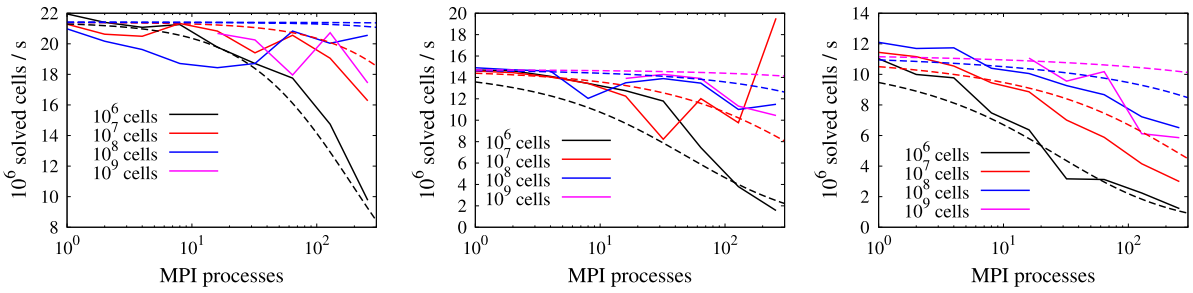

Fig. 1. Scalability test results of the OPar particle-in-cell code. The figures plot the number of grid cell updates per second per process against the number of MPI processes. Runs have been performed in 1d (left), 2d (middle) and 3d (right). Solid lines represent the measured data and dashed lines represent the results from the performance model.

Nonetheless, Schnek does have an impact on the performance of simulation codes because it supplies the data structures for the numerical grid and provides the mechanisms for passing data between processes in a parallel environment. In order to provide a benchmark, we will report on the performance of the FDTD code which has been outlined in the examples throughout this article. The code simulates Maxwell's equations on a Yee grid using periodic boundary conditions. Material can be added by specifying a constant relative permittivity ε_r globally. The complete code, including setup through the configuration file, MPI parallelisation, and data output into HDF5 files uses no more than 400 lines of C++ code.

As a reference run, a plane wave was set up with a wavelength of $\lambda = 1\ \mu$m. The boundary conditions of the simulation domain were periodic. The code was slightly modified to create 1d, 2d, and 3d versions. The one-dimensional runs were performed with 10^6, 10^7, 10^8, and 10^9 grid cells. For the two dimensional simulations, a square grid with sizes of 1000^2, 3162^2, 1000^2, and 31622^2 grid cells was simulated. The numbers are chosen so that the total number of cells is close to the numbers of cells in the one dimensional runs. Similarly, the number of cells in the three dimensional runs is 100^3, 215^3, 464^3, and 1000^3. The grid spacing Δx was chosen to be as large as possible but no larger than $\Delta x = 0.05\ \mu$m, while ensuring the size of the total simulation domain was a multiple of the wavelength λ. The time step Δt is determined by the Courant–Friedrichs–Lewy condition for the FDTD scheme, $\Delta t = \Delta x/c$ where = Δx is the grid spacing and c is the speed of light. Output files were written in initial runs to ensure the correctness of the simulation. In the benchmark runs no output files were written as to avoid benchmarking the I/O performance of the computing cluster.

Fig. 1 shows the results of the scalability test. The figures show the number of grid cell updates per second per process for different runs, depending on the number of MPI processes. Runs have been performed on 1 to 256 processes and all configurations show good scalability. The runs with the smallest number of grid cells (10^6 cells) show some deterioration as the processor number increases. This is expected as the ratio of ghost cells to inner cells increases with the number of processors and communication becomes more and more important. It can also be seen that this deterioration is worse in higher dimensions due to the larger surface to volume ratio as the dimension increases. We analyse these results using the following performance model. Let T_{comp} be the time used to compute the update of a single grid cell. This update is not performed within the Schnek library itself and thus cannot be controlled by the library. T_{comp} can vary by orders of magnitude depending on the numerical scheme implemented by the developer. For the FDTD scheme presented here as an example, T_{comp} will be comparatively small as each update contains only three subtraction, two multiplication, and three division operations, see listing 3. Other numerical schemes can have more than an order of magnitude more operations per cell update.

Let T_{comm} be the time taken for exchanging a single grid cell between processes, and T_{lat} the latency time for initiating a data exchange. Assuming the simulation contains a single grid, we then obtain for the time taken for a single time step,

$$T_{\text{step}} = n_L T_{\text{comp}} + N_{\text{comm}} \left(n_B T_{\text{comm}} + T_{\text{lat}}\right). \tag{1}$$

Here n_L is the total number of grid cells per process, n_B is the number of ghost cells exchanged during a single exchange, and N_{comm} is the number of exchange operations per time step. Assume that the simulation domain is a d-dimensional cube with l_G grid cells along each side, $n_G = l_G^d$. The domain is decomposed into a d-dimensional cubic array of MPI processes, $n_P = l_P^d$. Each MPI process then contains

$$n_L = \frac{n_G}{n_P} = \left(\frac{l_G}{l_P}\right)^d \tag{2}$$

grid cells. The number of boundary cells per process in each dimension is then given by

$$n_B = g\left(\frac{l_G}{l_P}\right)^{d-1} = g\left(\frac{n_G}{n_P}\right)^{\frac{d-1}{d}}, \tag{3}$$

where g is the thickness of the ghost cell layer. Each local domain has $2d$ ghost cell borders which are communicated individually, $N_{\text{comm}} = 2d$. We then find

$$T_{\text{step}} = N_f \frac{n_G}{n_P} T_{\text{comp}} + N_f 2d \left(g\left(\frac{n_G}{n_P}\right)^{\frac{d-1}{d}} T_{\text{comm}} + T_{\text{lat}}\right). \tag{4}$$

Here we have included an additional factor N_f to account for the number of individual fields that have to be exchanged. For the FDTD algorithm described here, we have $N_f = 6$ and $d = 1, 2, 3$. Eq. (4) allows comparison with the results from the scaling runs. T_{comm} can be estimated from the single core runs by ignoring the contributions due to communication. The other two parameters, T_{comp} and T_{lat}, have been obtained by visually matching the curves with the data in Fig. 1. For the 1d case we find $T_{\text{comp,1d}} \approx 7.8 \times 10^{-9}$ s and

$T_{\mathrm{comm,1d}} \approx 5 \times 10^{-6}$ s. Note that in the 1d case, T_{lat} cannot be determined independently as it possesses the same scaling as T_{comm}. For the 2d case we find $T_{\mathrm{comp,2d}} \approx 1.1 \times 10^{-8}$ s, $T_{\mathrm{comm,2d}} \approx 10^{-7}$ s and $T_{\mathrm{lat,2d}} \approx 2.5 \times 10^{-4}$ s, and for the 3d case $T_{\mathrm{comp,3d}} \approx 1.5 \times 10^{-8}$ s, $T_{\mathrm{comm,3d}} \approx 2 \times 10^{-8}$ s and $T_{\mathrm{lat,3d}} \approx 5 \times 10^{-4}$ s. The corresponding curves have been plotted in Fig. 1 as dashed lines for comparison. We remark that the values given here are obtained from a very small data set, and thus contain large numerical uncertainty. In addition T_{comm} and T_{lat} depend on the computer architecture used. We stress again that T_{comp} will strongly depend on the numerical algorithm implemented by the developer. The example shown here involved only a few mathematical operations on each grid cell per time step. For other algorithms T_{comp} can increase by orders of magnitude, thus drastically improving the scaling behaviour of the code.

7. Summary

Schnek is a C++ library that is intended to make the task of programming a large-scale parallel simulation easier. It is designed to use regular grids although some of its features can be useful for any type of simulation code. The library defines class templates for storing and manipulating regular grids. A specialisation of the `Grid` class includes information about the physical extent of the grid and the grid stagger and the `MPICartSubdivision` class allows the grids to be distributed over multiple processors. Schnek provides an interface to read setup files and fill simulation parameters and initial conditions from values in the setup file. The syntax of the file parser allows the use of mathematical expressions and user defined variables. The ability to store expressions for later evaluation makes it possible to use user defined formulas to provide space or time dependent input parameters. The setup file is used to construct a hierarchical structure of modules in a simulation run. It is easy for developers to define modules that can be selected by simple keywords in the setup file. In order to encourage collaborative code development each module can provide literature references. These can be used to create a BibTex file with references and a LaTeX file with the descriptions of the modules used.

Schnek provides a large part of the framework that is needed to write simulation codes. In a practical use case the main part of the computations is performed by the software that uses the library. For this reason it is not meaningful to benchmark Schnek directly. We can, however, use a representative simulation code to show that Schnek does not introduce any bottlenecks that might prevent codes from scaling onto a large number of processors. To this end we have used the full FDTD code outlined in the examples throughout this article. We have shown that the scaling of the code is very good up to 256 processes. Only in the cases where the communication between the processes is naturally expected to take up a large part of the work does the scaling degrade.

The library has been designed to be both powerful and easy to use. One goal was to ensure portability by reduction of dependencies on other libraries. Schnek make heavy use of the Boost library, which is now a de-facto standard across many architectures. The dependencies on the MPI and HDF5 libraries, while necessary for some features of Schnek, are optional. The code deliberately does not make use of any features of the C++11 standard in order to improve compatibility with a wide range of compilers.

Acknowledgements

The author would like to thank Raoul Trines for his assistance in preparing this document. This work is partly funded by the European Research Council, grant STRUCMAGFAST (Project ID: 307238, Funded under: FP7-IDEAS-ERC). The author is grateful for the use of computing resources provided by STFC's Scientific Computing Department.

References

[1] TOP500 Supercomputer Sites, 2016. http://www.top500.org/list/2016/11.
[2] J. Leng, W. Sharrock, in: J. Segal, C. Morris (Eds.), Handbook of Research on Computational Science and Engineering: Theory and Practice, IGI Global, 2012, pp. 177–196. http://dx.doi.org/10.4018/978-1-61350-116-0.ch008.
[3] B. Stroustrup, The C++ Programming Language, Addison-Wesley, 1997.
[4] J. Cary, S. Shasharina, J.C. Cummings, J.V.W. Reynders, P.J. Hinker, Comput. Phys. Comm. 105 (1997) 20–36.
[5] T.S. Grigera, Comput. Phys. Comm. 182 (10) (2011) 2122–2131.
[6] T.L. Veldhuizen, in: D. Caromel, R.R. Oldehoeft, M. Tholburn (Eds.), Computing in Object-Oriented Parallel Environments, in: Lecture Notes in Computer Science, Springer, Berlin Heidelberg, 1998, pp. 223–230.
[7] J.G. Siek, A. Lumsdaine, in: D. Caromel, R.R. Oldehoeft, M. Tholburn (Eds.), Computing in Object-Oriented Parallel Environments, in: Lecture Notes in Computer Science, Springer, Berlin Heidelberg, 1998, pp. 59–70.
[8] I. Honkonen, S. von Alfthan, A. Sandroos, P. Janhunen, M. Palmroth, Comput. Phys. Comm. 184 (4) (2013) 1297–1309.
[9] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, L. Rauchwerger, Lang. Compilers Parallel Comput. (2003) 193–208.
[10] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, 1999.
[11] M. Folk, A. Cheng, K. Yates, Proceedings of Supercomputing, 1999.
[12] S. Balay, S. Abhyankar, M. Adams, P. Brune, K. Buschelman, L. Dalcin, W. Gropp, B. Smith, D. Karpeyev, D. Kaushik, Petsc users manual revision 3.7, Tech. rep, Argonne National Lab. (ANL), Argonne, IL (United States), 2016.
[13] M.A. Heroux, R.A. Bartlett, V.E. Howle, R.J. Hoekstra, J.J. Hu, T.G. Kolda, R.B. Lehoucq, K.R. Long, R.P. Pawlowski, E.T. Phipps, ACM Trans. Math. Softw. 31 (3) (2005) 397–423.
[14] R.D. Falgout, U.M. Yang, International Conference on Computational Science, 2002, pp. 632–641.
[15] H.C. Edwards, C.R. Trott, D. Sunderland, J. Parallel Distrib. Comput. 74 (12) (2014) 3202–3216.
[16] C.K. Birdsall, A.B. Langdon, Plasma Physics via Computer Simulation, McGraw-Hill, New York, 1985.
[17] B. Dawes, D. Abrahams, R. Rivera, Boost C++ libraries. http://www.boost.org.
[18] H. Schmitz, holgerschmitz (Holger Schmitz) / Repositories. https://github.com/holgerschmitz?tab=repositories.
[19] K. Yee, IEEE Trans. AP-14 (1966) 302.
[20] M. Frigo, S.G. Johnson, Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on, 1998, pp. 1381–1384.
[21] T.L. Veldhuizen, C++ Rep. 7 (5) (1995) 26–31.
[22] T.D. Arber, K. Bennett, C.S. Brady, A. Lawrence-Douglas, M.G. Ramsay, N.J. Sircombe, P. Gillies, R.G. Evans, H. Schmitz, a.R. Bell, C.P. Ridgers, Plasma Phys. Control. Fusion 57 (11) (2015) 113001.
[23] R.A. Fonseca, L.O. Silva, F.S. Tsung, V.K. Decyk, W. Lu, C. Ren, W.B. Mori, S. Deng, S. Lee, T. Katsouleas, International Conference on Computational Science, 2002, pp. 342–351.
[24] F. Hecht, J. Numer. Math. 20 (3–4) (2012) 251–265.
[25] A. Thurston, Ragel state machine compiler. http://www.colm.net/open-source/ragel.
[26] D.R. Hipp, The LEMON Parser Generator. http://www.hwaci.com/sw/lemon/.

[27] J. Corbet, G. Kroah-Hartman, A. McPherson, Linux Kernel Development, Tech. Rep. September, The Linux Foundation, 2013. http://www.linuxfoundation.org/publications/linux-foundation/who-writes-linux-2013.
[28] S. Weber, The Success of Open source, Harvard University Press, 2005.