MMSP

The Mesoscale Microstructure Simulation Project

Jason Gruber

October 31, 2009

Contents

Chapter 1

Introduction

The goal of the Mesoscale Microstructure Simulation Project (`MMSP`) is to provide a simple, consistent, and extensible programming interface for all grid and mesh based microstructure evolution methods. Simple means that the package has a very small learning curve, and for most routine simulations, only a minimal amount of code must be written. By consistent we mean, for example, that code for two-dimensional simulations is nearly identical to that for three-dimensional simulations, single processor programs are easily parallelized, and fundamentally different methods like Monte Carlo or phase field have the same look and feel. Finally, extensible means that it's straightforward to add new grid types or physical behaviors to the package. Other considerations include efficiency and portability (`MMSP` is written entirely in ISO compliant `c++`).

1.1 The `MMSP` concept

The design of `MMSP` is based on several observations about how mesoscale simulations are used by materials scientists: most mesoscale simulations discretize the spatial domain using a rectilinear grid. A data structure is associated with each grid node that has a particular size (scalar, vector, etc.) and value type (integer, floating point) depending on the simulation method. Most simulations update the data structure at each node in a way that falls under some common methodology (Monte Carlo, phase field, etc.) but has features unique to each given physical process. What this roughly means is that most mesoscale simulations use a common spatial discretization, but we usually need to tweak the details of how we represent spatial data and how we update it. The unfortunate truth is that typically, a researcher wishing to model a particular physical process produces code with a focus mainly on the particulars of the process itself (the "tweak"), largely ignoring the problem of how to design reusable data structures. What happens when they decide they should try a different simulation method, or when they realize that they need to use a parallel implementation? It then becomes apparent that more flexible data structures should have been used in the first place. The purpose of `MMSP` is to provide the core functionality that we don't necessarily want to think about each time we program a new method. `MMSP` helps keep its users from reinventing grid data structures, file input and output, parallelization, handling boundary conditions, etc. while retaining enough flexibility to model a large number of physical processes.

Those familiar with similar code packages might have already noticed that the `MMSP` concept is a bit unusual. Other packages typically provide a very high level interface intended for use with a *single* computational method. Using the interface typically means learning package-specific methodology, classes, functions, methods, etc. In contrast, `MMSP` is meant to be used for any and all grid-based methods, and provides a much lower level interface. This results in a lot more flexibility in what `MMSP` can do. And while `MMSP` still requires some learning, users will find that they are able to leverage much more of their previous programming experience.

1.2 What `MMSP` does

`MMSP` is nothing more than a collection of `c++` header files that declare a number of `grid` objects (classes) and define how most of their methods (member functions) are implemented. Some things `MMSP` provides include:

- A simple, extensible programming interface
- Computational grids of arbitrary dimension
- Parallel implementations using MPI
- Automatic, optimal parallel mesh topologies
- Utility programs for grid visualization
- Classes for Monte Carlo methods
- Classes for cellular automata methods
- Classes for phase field methods (conventional)
- Classes for phase field methods (sparsePF)
- Classes for general finite difference PDE solvers
- Example simulation methods and grid objects

1.3 What `MMSP` doesn't do

`MMSP` is not the kind of software that reads a few parameters or an input file specified by the user and cranks out some generic computation. In fact, `MMSP` relies on the user to provide code for all of the real physics that the simulation is intended to capture. This isn't as scary as it sounds. `MMSP` was designed to make this procedure as simple as possible. The takeaway message here is that `MMSP` makes programming materials simulation code easier, but it isn't a "black box" that can be used by a complete novice.

1.4 What `MMSP` requires

The `MMSP` interface is intended to look and feel very natural for most programmers with experience in scientific computing. While many of the most advanced features of `c++` have been used in creating the grid and data classes, the user need not be proficient in anything other than basic procedural programming. `Fortran`, `c`, and novice `c++` programmers alike will find that `MMSP` is quite easy to use. Basic requirements include:

- Minimal programming experience
- A `c++` compiler with ISO compliant libraries and headers (e.g. `gcc` 2.95 or later)
- MPI libraries are required if compiling parallel programs (e.g. OpenMPI)

1.5 Terms of use

`MMSP` is freely available for anyone performing non–profit scientific research; those interested in using `MMSP` for any other purposes should contact the author. We give no guarantees whatsoever about the capabilities of `MMSP`. If you use `MMSP` in your research, please tell others about it, send us any new code you'd like to see incorporated into the package, and above all, give us feedback!

Chapter 2

Getting started with MMSP

The following sections present a few short tutorials on writing, compiling, and running simple MMSP programs. For those totally confused by the syntax of the examples presented here we suggest consulting an introductory c++ text.

2.1 A quick tutorial

2.1.1 The Hello MMSP! program

Because every good programming language or interface tutorial starts with a "Hello World!" example program, we'll do the same. For most MMSP applications, we include a header file named MMSP.hpp. Then we need a main() program and a few lines to print out our message. Here it is:

```
#include"MMSP.hpp"

int main(int argc, char* argv[])
{
    MMSP::Init(argc,argv);

    std::cout<<"Hello MMSP!"<<std::endl;

    MMSP::Finalize();
}
```

The only code here that should look out of the ordinary are the statements

```
MMSP::Init(argc,argv);
```

and

```
MMSP::Finalize();
```

What do these lines do? For single processor programs, they do absolutely nothing – they could actually be removed without any consequences. However, for programs that use the message passing interface (MPI), they act as wrappers for the similarly named `MPI::Init` and `MPI::Finalize` commands. It's useful to include them here because they'll allow us to write programs that may be compiled for both single or multiple processor environments.

Programmers familiar with `c++` will notice that there's obviously some `MMSP` namespace being used here. For those less familiar, namespaces are a somewhat recent addition to `c++` that are used as a means of avoiding naming conflicts. We can avoid using namespace resolution so frequently if we use the relevant `using` statement, e.g.

```
#include"MMSP.hpp"
using namespace MMSP;

int main(int argc, char* argv[])
{
    Init(argc,argv);

    // etc.
}
```

Namespaces serve to prevent programming errors and to ensure code reusability, so naturally we should use `using` statements with care.

The observant reader may also notice that we've used some of the stream input/output functions of `c++` without including the requisite `<iostream>` header. In fact, this header and many other standard `c++` headers are included implicitly thorough the file `MMSP.hpp`. If you need a particular standard header for your application and aren't sure if it has been included by `MMSP.hpp`, you can always just `#include` it in your source code without any ill effects.

So that's it as far as the code goes. This is a source file that may be compiled for both single and parallel simulations of ... nothing. In a moment we will move on to code that actually does something, but for now we should say a few words about compiling the code and running the executable.

2.1.2 Compiling and running `Hello MMSP!`

Compiling code is a task that is, unfortunately, fairly platform dependent. While `MMSP` programs should compile easily on any platform, the required steps to do so may not look like the method shown here. That said, let's look at how we would compile the previous example for a typical Linux or Unix setup. Suppose the above code has been saved to a file named `hello.cpp`, and that `MMSP` has been extracted to the same directory. A typical Linux machine will have at the least the GNU family of compilers installed, in which case we would type the command

```
g++ -I MMSP/include hello.cpp -o hello
```

which produces an executable file named `hello`. The compiler option `-I ...` suggests a directory to search for non-standard headers; if your MMSP headers live somewhere else, you'll need to make the appropriate change. To run the program, we type

```
    ./hello
```

which should produce our message. Many other compilers on Linux and Unix machines use the same options as listed here, so this line may be very close to what you would use, even if you're not using `gcc`.

Now let's move on to parallel compilation. We assume that if you're not skipping over this part, you have the MPI libraries installed on your machine (as well as your cluster). With a typical MPI installation, a number of programs are included which effectively wrap your usual `c++` compiler with a script named something like `mpic++`, `mpicxx`, `mpicc`, etc. With this in mind, we issue a command which may look like

```
    mpicxx -I MMSP/include -include mpi.h hello.cpp -o hello
```

which again produces an executable named `hello`. This time, however, we need to run `hello` using an MPI command such as `mpirun` or `mpiexec` (see the documentation for your MPI distribution),

```
    mpirun -np 4 hello
```

which in this case produces our message four times. The author sincerely hopes that your experience is this straightforward, but don't count on it!

2.2 An example using the `grid` class

In this section, we look at an example that uses the MMSP `grid` template class, and actually resembles part of a program you might actually use:

```
    #include"MMSP.hpp"
    using namespace MMSP;

    int main(int argc, char* argv[])
    {
        Init(argc,argv);

        grid<2,int> GRID(1,0,100,0,100);

        for (int x=xmin(GRID); x<xmax(GRID); x++)
            for (int y=ymin(GRID); y<ymax(GRID); y++)
                GRID[x][y] = x+y;

        output(GRID,"testgrid");

        Finalize();
    }
```

This program performs the amazing feat of generating a new 2-dimensional `grid` object, assigning to each node the sum of `x` and `y`, and then writing the final state of the `grid` to a file, the name of which is simply `testgrid`. Of course this isn't really anything special, but the imaginative reader may begin to see how this program might be used as a template to generate other, more useful `grid` objects.

Several new features have been introduced in this example. First, the line

```
    grid<2,int> GRID(1,0,100,0,100);
```

constructs the `grid` object. Objects of the `grid` class take two template arguments, the first being the dimension (`2`) and the second being the data type (`int`). In `MMSP`, the dimension can be any integer greater than zero, while the data type can be any of `c++`'s built-in types or any of the `MMSP` data types to be discussed later. The name of the `grid` comes next (`GRID`), followed by constructor arguments in parentheses. These arguments indicate the total number of fields to create at each node (relevant only for vector-like data types) and the lower/upper limits in the `x` and `y` directions, respectively. Note that these limits are the *global* limits, i.e. the local grid size will be smaller in a parallel program. Also note that the limits may be either positive or negative, as long as the upper limit is always larger than the lower limit. To give a sense of how the above generalizes to other dimensions and data types, consider the following,

```
    grid<3,double> GRID(1,0,128,0,256,0,256);
```

which would produce a 3-dimensional `grid` object with size $128 \times 256 \times 256$ and the built-in data type `double`.

The next lines iterate through the nodes of the `grid`,

```
    for (int x=xmin(GRID); x<xmax(GRID); x++)
        for (int y=ymin(GRID); y<ymax(GRID); y++)
            GRID[x][y] = x+y;
```

The functions `xmin`, `xmax` and `ymin`, `ymax` have been invoked to retrieve the *local* limits in each direction. As suggested above, the local grid size in a parallel program is always smaller than the global grid size because parallelization in `MMSP` is achieved through spatial subdivision. Use of these functions ensures that we iterate only through the nodes stored by the local process. While it isn't strictly necessary to use these functions in code meant to be run on a single processor, using them now will make parallelization completely trivial.

Also of note here is the fact that subscripting works the same way for `MMSP` grid objects as it does for subscripted arrays. However, we will later see that the `MMSP` subscript operators are "smarter" in the sense that they're cognizant of boundary conditions and adjust appropriately for calls made to nodes that are "out of bounds."

Finally, the line

```
    output(GRID,"testgrid");
```

seems fairly self-explanatory, and it is. We note, however, that in the case of a parallel program, the `output` function performs the additional task of pieceing back together the global `grid` object from all local `grid` objects before writing to the file.

As a point of reference, let's now look at how we might do something similar to the example above with the usual `c` or `c++` subscripted arrays:

```
int main(int argc, char* argv[])
{
    int GRID[100][100];

    for (int x=0; x<100; x++)
        for (int y=0; y<100; y++)
            GRID[x][y] = x+y;

    // etc.
}
```

This may look considerably simpler, but consider the following: how do we run this in parallel? The immediate answer is that we introduce MPI funtion calls, but then we also need to decide how to subdivide the grid, how to deal with boundary conditions, how to put the pieces back together when we're done, etc. And what if we want to change the number of processors or number of nodes in each direction and maintain an optimal subdivision scheme? All of these things are done by `MMSP` automatically.

2.3 The prototypical MMSP program

MMSP was created with the intent that it would be used for mesoscale microstructure simulation. So let's look at how we would typically write code to do just that.

```
// prototype.cpp
#include"update.hpp"
using namespace MMSP;

int main(int argc, char* argv[])
{
    Init(argc,argv);

    grid<2,double> GRID(argv[1]);

    update(grid,atoi(argv[3]);

    output(GRID,argv[2]);

    Finalize();
}
```

In this example, in addition to the usual MMSP boilerplate code, we have a grid constructor that reads from a file with a name specified by argv[1], a call to some function called update (more on this in a moment), and a call to output the grid object to a file with a name specified by argv[2]. After compiling this program, we would run it with the command line

```
./program initial.PF final.PF 100
```

or, if we compiled with the MPI libraries, we would run in with something similar to

```
mpirun -np 4 program initial.PF final.PF 100
```

Here, 100 is the intended number of time steps that the update function will perform.

Now that we have our prototype main() function, let's look at the update function. For this example, we would have a new header file named update.hpp,

```
// update.hpp
#include "MMSP.hpp"
using namespace MMSP;

void update(grid<2,double>& GRID, int steps)
{
    grid<2,double> update(GRID);

    for (int step=0; step<steps; step++) {
        for (int x=xmin(GRID); x<xmax(GRID); x++)
            for (int y=ymin(GRID); y<ymax(GRID); y++) {

                // replace with "real" computations...
                update[x][y] = GRID[x][y];
            }

        swap(GRID,update);
        ghostswap(GRID);
    }
}
```

Here we define the `update` function, with its first argument a `grid` object and its second argument the number of time steps to perform. The only thing we should bring special attention to here is the use of the ampersand (`&`) to force call-by-reference, which overrides `c++`'s usual convention of call-by-value (see a typical introductory text for more).

On the first line within the `update` function, we create a new `grid` object, itself called `update`. The new `grid` is created using a constructor with `GRID` as its argument. What this does is it generates a `grid` with the same spatial extent, number of fields, parallel topology, etc. In other words, the `grid` called `update` is a suitable workspace to store the values of `GRID` for the next time step as we compute them.

Next, we have a loop over all time steps. Within this loop we iterate through the nodes of `GRID` just as in the example of the previous section. Again, note that the limiting values of `x` and `y` are obtained by appropriate function calls. At each node, we would normally perform some meaningful computation to determine the value of `GRID` at the next time step, then store the value in `update`. Here we simply copy the value of `GRID` at this node to `update`.

After computations are performed at each node, we have two more operations. First, we `swap` the data of `GRID` and `update`. `GRID` now contains the new values, while `update`, the workspace, contains the old. Next, a function called `ghostswap` is called with `GRID` as its argument. For single processor programs, the `ghostswap` function does nothing. For parallel programs, it performs a coordinated series of “ghost” cell data swaps, such that the ghosts associated with the local portion of `GRID` are filled with the appropriate data from all neighboring processors. And again, while it’s not critical to include this line when our intent is to write a single processor program, it will serve us well to include it here because it will allow us to easily parallelize the code later.

Chapter 3

`MMSP` data classes

3.1 Introduction

Aside from the dimension, the other defining characteristic of a `grid` object is the type of data stored at each node or cell. In `MMSP`, valid `grid` data may be either built-in variable types or one of several `MMSP` data classes. This chapter presents a more detailed look at using both kinds of data types.

3.2 Common features

`MMSP` data classes share a number of common features. Most importantly, the way we interact with them through functions and symbolic operators has been designed to be as intuitive and self-consistent as possible. In this section, we list the functions common to all `MMSP` data classes and built-in types, and in following sections we describe additional functions specific to each particular class.

The member functions common to every `MMSP` data class include

- `length()`
  This function is used to determine the "length" of a given data object. For example, the length of a `scalar` is always one, the length of a `vector` object is the number of values stored in it, and so on.

- `resize(`*integral_value*`)`
  This function is used to "resize" data types that contain more than one value, such as a `vector` object. When an object inherently has length one (e.g. `scalar` objects), this function does nothing at all.

- `copy(`*data_object*`)`
  The `copy` function copies all data from the object in the parameter list to the calling object, i.e. the one that called the function.

- `swap(`*data_object*`)`
  The `swap` function swaps all data of the object in the parameter list with the calling object.

- `buffer_size()`
  This function returns the number of bytes that would be used if the calling object were stored in a character buffer.

- `to_buffer(`*character_buffer*`)`
  This function packs the data of the calling object into the character buffer object given in the parameter list. It also returns the number of bytes that were used.

- `from_buffer(`*character_buffer*`)`
  This function reads data from the character buffer object given in the parameter list and writes it to the calling function. It also returns the number of bytes that were read.

- `read(`*ifstream_object*`)`
  The `read` function reads data from the given `c++ ifstream` object to the calling data object.

- `write(`*ofstream_object*`)`
  The `write` function writes the data of the calling object to the the `c++ ofstream` object given in the parameter list.

- `operator=(`*data_object*`)`
  The assignment operator is overloaded for all `MMSP` data classes to have the usual “copy” behavior.

Each of these functions may be called in either of two ways: as a class member function or as a globally defined function. In the first case, calls to a member function `f(...)` look like this:

data_object`.f(`*parameter_list*`);`

In the second case, the global function `f` is called with a similar syntax, but now the data object becomes the first function parameter:

`f(`*data_object*`,` *parameter_list*`);`

Globally defined functions have the same behavior as their associated grid member function. The user is free to choose either based on convenience or aesthetics. Symbolic operators are the only member functions that do not have associated functions.

It is, of course, also possible to use built-in data types such as `int`, `float`, and `double` anywhere an `MMSP` data object might be used. The functions listed above have been redefined to work for all built-in types, but note that because it’s impossible to write member functions for built-ins, only the global function calls may be used.

3.3 Using MMSP data types

All MMSP data types are defined as template classes. Just as a declaration of a grid objects requires dimension and data type template parameters, each MMSP data class object requires a single template parameter.

Most of the time, the user won't work with MMSP data types directly, but through nodal data on grids. It's important to know some of the basic workings of the MMSP data classes for this reason.

3.3.1 The scalar class

The scalar class is essentially a wrapper for built-in data types. It was defined primarily for consistency, i.e. as a complement to the
vector class discussed below. MMSP does not define any functions for scalar objects other than the common functions discussed in the previous section. However, because a scalar object is an instantiation of a class, it is possible to use both the member function and global function calling syntax.

```
// MMSP scalar example

// declaration of a "scalar" object
scalar<double> s;

// use a "scalar" as you would use the corresponding built-in
s = 1.2345;
double square = s*s;
std::cout<<"s = "<<s<<std::endl;

// the following two lines are equivalent
// and both write "1" to standard output
std::cout<<"length = "<<length(s)<<std::endl;
std::cout<<"length = "<<s.length()<<std::endl;

// how a "scalar" is used in a "grid"
grid<2,scalar<int> > GRID(1,0,10,0,10);
for (int x=xmin(GRID); x<xmax(GRID); x++)
    for (int y=ymin(GRID); y<ymay(GRID); y++)
        GRID[x][y] = x+y;

// this writes the value "1" to standard output
std::cout<<"fields = "<<fields(GRID)<<std::endl;
```

3.3.2 The vector class

The vector class is meant to be used primarily as a fixed-length container for nodal data. Just like a usual c or c++ array, vector data is accessed by use of the subscript operator.

When a `vector` is used as the data associated with a `grid` node, its length is initialized to the number of fields assigned to the `grid`. Otherwise, a `vector`'s length must be initialized with the `resize` funtion.

If a `vector` of nonzero length is `resize`d to a greater length, the original data it contains is preserved. Thus a `vector` may be used, e.g. to a generate list even when its final length is not known *a priori*. Trying to `resize` a `vector` being used as `grid` data is *not* reccomended.

```
// MMSP vector example

// declaration of a "vector"
vector<double> v;

// a "vector" must be resized before use
resize(v,10);

// the following two lines are equivalent
// and both write "10" to standard output
std::cout<<"length = "<<length(v)<<std::endl;
std::cout<<"length = "<<v.length()<<std::endl;

// set and get values with the subscript operator
v[0] = 1.2345;
for (int i=1; i<length(v); i++)
    v[i] = 2.0*v[i-1];

// how a "vector" is used in a "grid"
grid<2,vector<int> > GRID(4,0,10,0,10);
for (int x=xmin(GRID); x<xmax(GRID); x++)
    for (int y=ymin(GRID); y<ymay(GRID); y++)
        for (int i=0; i<fields(GRID); i++)
             GRID[x][y][i] = x+y+i;

// this writes the value "4" to standard output
std::cout<<"fields = "<<fields(GRID)<<std::endl;
```

3.3.3 The sparse class

The MMSP `sparse` data class contains `vector`-like data, but rather than maintaining a fixed length of data, `sparse` objects grow as values are `set` by the user. `sparse` data is most useful in situations where a large number of fields are defined, but only a small number of them have values that differ from some "nominal" value.

```
    // MMSP sparse example

    // declaration of a "sparse"
    sparse<double> s;

    // the following two lines are equivalent
    // and both write "0" to standard output
    std::cout<<"length = "<<length(s)<<std::endl;
    std::cout<<"length = "<<s.length()<<std::endl;

    // all of the following lines
    // write "0" to standard output
    for (int i=0; i<10; i++)
        std::cout<<"s["<<i<<"] = "<<s[i]<<std::endl;

    // use of the "set" function
    set(s,0) = 1.2345;
    set(s,1) = s[0];
    set(s,2) = s[1];

    // now the length is "3" ...
    std::cout<<"length = "<<length(s)<<std::endl;

    // ... and we can output only those values that are stored
    // Note that both calls to "cout" produce the same output
    for (int i=0; i<length(s); i++) {
        int ind = index(s,i);
        double val = value(s,i);
        std::cout<<"s["<<ind<<"] = "<<val<<std::endl;
        std::cout<<"s["<<ind<<"] = "<<s[ind]<<std::endl;
}

    // how a "sparse" is used in a "grid"
    grid<2,vector<int> > GRID(0,0,10,0,10);
    for (int x=xmin(GRID); x<xmax(GRID); x++)
        for (int y=ymin(GRID); y<ymay(GRID); y++)
            for (int i=0; i<10; i++)
                 set(GRID[x][y],i) = x+y+i;
```

The `sparse` data class achieves its function by storing index/value pairs as they are `set` by the user. Obviously, this requires more memory than storing a single value, so that there's some point where, for a small enough number of fields, using `sparse` data becomes impractical.

The subscript operator applied to `sparse` data returns the value associated with the index given as the operator's argument. If the value for the given index has not been `set` by the user, the subscript operator returns zero.

The `length` function applied to a `sparse` object returns the number of values that have been `set`. The user may iterate through only those values that have been `set` by using the `value` and `index` functions. A synopsis of the functions unique to the `sparse` data type follow.

- `set(`*integral_value index*`)`
  This function set the value of a particular index/value pair of a `sparse` object.

- `value(`*integral_value i*`)`
  This function returns the value of the i^{th} index/value pair of a `sparse` object.

- `index(`*integral_value i*`)`
  This function returns the index of the i^{th} index/value pair of a `sparse` object.

3.4 Using built-in data types

3.5 Writing new data classes

Chapter 4

The grid class

The MMSP base class grid is defined in the header MMSP.grid.hpp. The derived template classes grid1D, grid2D, and grid3D provide the base for the high level grid objects associated with various simulation methods. The template parameters for each grid object are a point data structure and a fundamental numeric type. The grid objects contain a large number of inherited methods (member functions) that perform common tasks such as file I/O, data storage and retrieval, etc. Member data for the grid classes determine the grid size, boundary conditions and point spacing along each coordinate axis. High level grid objects usually have a name that indicates both their intended use as well as their dimensionality, e.g. PFgrid3D and MCgrid2D. These are the grid objects that are used in performing simulations or other computations. Detailed information about each existing grid class is given in the reference section of this page. See also the source distribution. MMSP grid classes are designed so that programming with them is as intuitive as possible. For example, data can be accessed by subscript operators in exactly the same way we would use data stored in a subscripted array // First create a grid object. PFgrid2D grid(100,100,2);

// Assign a value to the phase field grid. grid[10][20][1] = 1.0;

// Use a value from the grid in a computation. float value = grid[10][20][1]; Because of this, C/C++ code that performs computations using "hard-coded" arrays automatically works for the corresponding MMSP grid object. This makes it trivial to insert MMSP grid objects into existing grid generation, simulation, or analysis code. Each grid object also provides a neighbor() function that returns a grid data structure using coordinates relative to a given position (x,y[,z]) // First create a grid object. MCgrid2D grid(100,100);

// Assign a value to the Monte Carlo grid. grid[10][20] = 5;

// Access the value using the neighbor() function. int spin = grid.neighbor(9,22,1,-2); The neighbor function automatically accounts for the grid boundary conditions set by the user. See the next subsection for more details on programming with the grid classes. Finally, let's consider using grid objects in a parallel computing setting. In single processor programs, we typically create a single grid object within the main() function. This grid object is considered to be the entire simulation domain. In parallel programs, however, the grid object declared within main() is a subgrid of a larger, global simulation domain. MMSP can automatically decompose (or recompose) a global grid object into local subgrid ”slabs” with cuts perpendicular to the x axis, a method which has a consistent (and simple) implementation regardless of grid dimension. MMSP allows the user to chose the number of ghost cells retained in each subgrid, and the boundary conditions of the global grid object are satisfied when the neighbor() function is used. The implementation of parallel grid functionality is contained within a separate header MMSP.grid.parallel.hpp for the convenience of users without a local MPI distribution.

4.1 Introduction

The `grid` class is the base class for all higher level grid classes...

A number of points about the grid class and its member functions are useful in the following.

- Most grid functions may be called in either of two ways: as a class member function or as a global friend function. In the first case, calls to a grid member function `f(...)` look like this:

  grid_object`.f(`*parameter_list*`);`

  In the second case, the global friend function `f` is called with a similar syntax:

  `f(`*grid_object*`, `*parameter_list*`);`

  Global friend functions have the same behavior as their associated grid member function. The user is free to choose either based on convenience or aesthetics. Note that the grid object now becomes the first argument to the function, while the remainder of the parameter list is unchanged. The only functions that don't follow this convention are symbolic operators, e.g. the subscript operator:

  grid_object`[x][y] = `*rvalue*`;`
  lvalue `= `*grid_object*`[x][y][z][index];`

  Symbolic operators are grid member functions that do not have associated friend functions.

-
-
-

4.2 grid class member functions

4.2.1 Constructors

```
grid(int fields, int x0, int x1, ...);
   grid(int fields, int min[dim], int max[dim]);
   grid(const grid& GRID);
   grid(char* filename);
```

4.2.2 Subscripting

4.2.3 File I/O

4.2.4 Buffer I/O

4.2.5 Accessing grid parameters

```
int x0(const grid& GRID, int i);
   int x1(const grid& GRID, int i);
   int x0(const grid& GRID);
   int x1(const grid& GRID);
   int y0(const grid& GRID);
   int y1(const grid& GRID);
   int z0(const grid& GRID);
   int z1(const grid& GRID);
```

4.2.6 Setting grid parameters

4.2.7 Utility functions

4.2.8 Parallel communications

4.2.9 Multigrid functionality

4.3 grid class examples

4.3.1 Constructing a grid

```
#include"MMSP.grid.hpp"

int main(int argc, char* argv[])
{
    MMSP::Init(argc,argv);

MMSP::grid<1,double> GRID(1,0,100);

    MMSP::Finalize();
}
```

```
#include"MMSP.grid.hpp"

int main(int argc, char* argv[])
{
    MMSP::Init(argc,argv);

    char* filename = argv[1];

MMSP::grid<2,double> GRID(filename);

    MMSP::Finalize();
}
```

```
#include"MMSP.grid.hpp"

int main(int argc, char* argv[])
{
    MMSP::Init(argc,argv);

MMSP::grid<3,MMSP::vector<double> > GRID();

    MMSP::Finalize();
}
```

4.3.2 Initializing a `grid`

4.3.3 Setting up `grid` parameters

4.3.4 Accessing `grid` parameters

4.3.5 Reading to and writing from files

4.3.6 Using `grid` in a parallel program

Chapter 5

Specialized grid classes

5.1 Introduction

5.2 `CAgrid`

5.3 `FDgrid`

5.4 `MCgrid`

5.5 `PFgrid`

5.6 `sparsePF`