DOI: 10.14529/jsfi190403

Improving Reliability of Supercomputer CFD Codes on Unstructured Meshes

Andrey V. Gorobets[1], *Pavel A. Bakhvalov*[1]

© The Authors 2019. This paper is published with open access at SuperFri.org

The paper describes a particular technical solution targeted at improving reliability and quality of a highly-parallel computational fluid dynamics code written in C++. The code considered is based on rather complex high-accuracy numerical methods and models for simulation of turbulent flows on unstructured hybrid meshes. The cost of software errors is very high in large-scale supercomputer simulations. Reproducing and localizing errors, especially "magic" unstable bugs related with wrong memory access, are extremely problematic due to the large amount of computing resources involved. In order to prevent, or at least notably filter out memory bugs, an approach of increased reliability is proposed for representing mesh data and organizing memory access. A set of containers is proposed, which causes no overhead in the release configuration compared to plain arrays. At the same time, it provides throughout access control in the safe mode configuration and additional compile-time protection from programming errors. Furthermore, it is fully compatible with heterogeneous computing within the OpenCL standard. The proposed approach provides internal debugging capabilities that allow us to localize problems directly in a supercomputer simulation.

Keywords: CFD, supercomputer, unstructured mesh, data structure, MPI, OpenMP, OpenCL.

Introduction

The importance of software reliability for supercomputer simulations can hardly be overestimated. Software errors that appear in large-scale simulations involving thousands of processor cores cause the loss of many CPU hours and large labor costs. Those readers who deal with such simulations using in-house codes most likely have noticed that the work on debugging parallel applications (often in conditions of strict time limits and burning deadlines) is especially stressing and hard. If an error shows up in a large-scale simulation, it means that it has infiltrated through the quality assurance procedure based on numerous test cases and preliminary simulations on coarse meshes. Therefore, most likely, this is a rather insidious bug that will be difficult to catch. Such errors typically exhibit unstable behavior that makes reproduction and localization difficult. The immense amount of suffering from debugging parallel applications that the authors experienced motivated the present work, which is aimed at increasing reliability of supercomputer simulation software.

A way to represent mesh data in a computational fluid dynamics (CFD) code is proposed. It is designed primarily for simulations using static unstructured meshes, dynamic meshes with constant topology and moving meshes with sliding interfaces. It is assumed that the code has release and safe mode build configurations, both using compilation with full optimization. The latter enables low-level checks that can affect performance. The following requirements were imposed on a set of containers for mesh data:

- no observable overhead in the release configuration compared to plain arrays;
- tolerable overhead in the safe mode configuration (say 20–30%, not several times);
- full access control in the safe mode configuration;
- informative diagnostics that reports where exactly the error occurred in the code and in which container;

[1]Keldysh Institute of Applied Mathematics (RAS), Moscow, Russian Federation

- only plain arrays at the backend of containers that provide direct compatibility with linear algebra libraries and computing on massively-parallel accelerators, such as GPUs.

One way or another, containers and structures for the mesh data are represented in every CFD code. These components play an important role in large-scale supercomputing. However, implementation details are in most cases hidden and not available in the literature. Typically, articles describing CFD software provide only a brief description of the general approach (see [11], for instance). In order to look at implementation details, we can anatomize third-party open source codes and learn how the data structure is organized. But this usually takes some effort. For instance, we have studied the Nectar++ open source CFD code [2]. It operates with containers, which are template classes based on plain pointers accessed via plain integers. There is no extra error protection.

There are multiple scientific groups working on general-purpose mesh data frameworks for scientific computing. Among them, the SIGMA team from the Argonne National Lab with the MOAB library [10] and an array-based mesh data structure [5]. There are relevant works in the Los Alamos National Laboratory [6, 7] for complex multi-material and geophysical mesh-based applications, respectively.

Another representative example of open source C++ software for scientific computing using mesh methods can be found in [4]. A brief overview of the relevant software is also presented there. This platform, called INMOST, has rather complicated containers and data structure due to its high generality. This may lead to notable overhead compared to narrower application-specific implementations. Also the low-level representation of data storage appears to be too complex for using in GPU computing.

Further examples of data structures for scientific computing can be found in [1, 12]. Implementation details are described in these works, but the proposed approaches do not provide additional error protection.

In the present work, we propose a simple approach for organizing mesh data in C++ that gives us additional correctness checks while causing no overhead in the release configuration. The containers have only plain pointers (1D vectors) at the backend. This ensures direct compatibility with heterogeneous computing within the OpenCL standard.

The following sections provide implementation details and representative code fragments that can help readers implement the proposed approach or some particular ideas in their CFD codes. In the next section, we define what the basic data for a mesh method consist of. Section 2 is devoted to containers that store these data. In Section 3, we address some complications, such as combining cell-centered and vertex-centered approaches in one code. Section 4 covers parallel computing aspects. Details on implementation of runtime diagnostics and memory allocation are given in Section 5. Performance evaluation is given in Section 6. Finally, the results of the study are summarized and the conclusions are given in the last section.

1. Basic Data for a Mesh Method

For the sake of simplicity, let us consider an unstructured hybrid mesh with a constant topology (only coordinates of nodes may change). There are numerous kinds of mesh objects in the computational domain (we say a kind, not a type, to avoid confusion with data types stored in containers). For instance, in our codes we have more than 10 types of objects, including: volume mesh nodes, edges, elements, faces, cells; boundary nodes, edges, faces; sliding interface nodes, edges, faces.

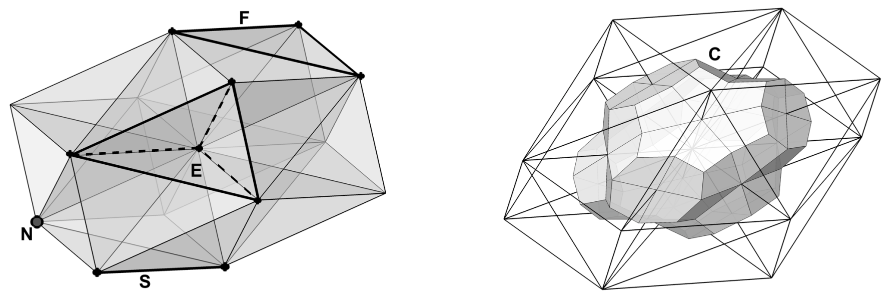
Note that there is a special kind, the set of cells (or control volumes, in other words), that introduces even more chaos and disorder. It switches between other kinds depending on where the mesh functions are defined. For vertex-centered schemes the set of cells is equal to the set of nodes, for element-centered schemes to the set of elements, respectively.

Furthermore, in computations we may need to know the list of adjacent objects of some kind for each object of the same kind or some other kind. It results in tens of so called topology containers that store this adjacency data. Since there are so many different kinds of mesh objects and relations between them, it is easy to make a mistake (and we often did it) by confusing types of objects when accessing containers.

To simplify the notation and explanation, let's limit ourselves just to the following few kinds of objects (denoted by one letter):

- **E** – mesh *E*lements;
- **N** – mesh *N*odes (vertices);
- **S** – mesh edges (in other words, line *S*egments);
- **F** – *F*aces of mesh elements.

Example of adjacent mesh objects is shown in Fig. 1.



(a) Tetrahedrons adjacent to one node (b) A vertex-centered control volume

Figure 1. Fragment of a tetrahedral mesh illustrating adjacent mesh objects

Then, let us denote different kinds of topology with two-letter notation, since for each object of the first kind the list of adjacent objects of the second kind is stored. The basic adjacency data comes from the main mesh topology (stored on disk):

- **EN** – for each mesh element stores the indices of its nodes.

The other topologies are derived from it:

- **EF** – for each element stores the indices of its faces;
- **FN** – for each face stores the indices of its nodes;
- **SN** – for each segment stores the indices of its nodes;

and, similarly, **ES**, **FS**.

Then, the following derivative adjacency data we can denote as the inverse topology:

- **NE** – for each node stores the indices of the elements it belongs to;
- **NS** – for each node stores the indices of the segments it belongs to;
- **FE** – for each face stores the indices of the elements it belongs to;

and, similarly, **NF**, **SF**, **SE**.

Then, there are adjacency data for elements of the same kind:

- **EE** – stores for each element the list of indices of the elements that share faces with this element;

- **NN** – for each node stores the list of indices of the nodes connected by an edge to this node.

This list is not complete. More topologies can be added for other combinations, if needed. Note that we have not considered the boundary surface so far, it will be addressed later.

Finally, data sets (mesh functions, structures of object properties, etc.) can be associated with the corresponding kinds of mesh objects or adjacency relations between them (such as non-zero entries of a sparse matrix that arise from the spatial discretization).

2. Basic Containers for Mesh Data

Since there is a chaos of numerous kinds of mesh objects and relations between them, some extra error protection mechanism would be very welcome. Basic index range check when accessing an array is insufficient in this case. In order to distinguish data associated with sets of objects of different kinds, we need a list of all possible kinds, an index that knows its kind, and an array that can be accessed only with an index of the proper kind.

In our simplified representation (Fig. 2), the list of possible kinds of mesh objects is given by the enumeration *tIdxKind*. The index class that knows its kind is just a wrapper around an integer value (can be 32 or 64 bits). The template class *tIdx* supplements our index with a kind using the *tIdxKind* enumeration. In doing so, an index of some kind can be used to access only arrays of the same kind. Violation of the kind will result in error at compilation time.

```
typedef int tInt; // Integer index value (to switch 32/64 bit if needed)

// Kinds of mesh elements
enum tIdxKind{ IDX_N=0/*nodes*/,IDX_E=1/*elements*/,IDX_S=2/*segments*/,IDX_F=3/*faces*/};

template<tIdxKind I> class tIdx { // Index of a given kind - wrapper around integer value
    tInt i; // Index value
    inline tIdx(const tInt j){ i=j; } // private direct cast from int
public:
    inline tInt idx() const { return i;}
    inline tIdx(){ i=-1; } // initialization with an incorrect value by default
    inline tIdx<I>& operator=(const tInt j){     i=j;   return *this; }
    inline tIdx<I>& operator=(const tIdx<I>& j){ i=j.i; return *this; }
    inline tIdx<I> operator+(tInt j) {  return (i+j); }
    inline tIdx<I>& operator++() { ++i; return *this; }
//  ... whatever other necessary functions and operators
};

// Error processing
int Crash(const char *fmt,...); // (MPI)Abort with message to log file, stdout and stderr
#ifdef SAFE_MODE // Safe assertion - to be placed in performance-critical places
  #define SAFE_ASSERT(X,...) if(!(X)) exit(Crash(__VA_ARGS__)); //enabled in safe-mode only
#else                        // Via exit – informs compiler/code analyzer about exit point
  #define SAFE_ASSERT(X,...) // disabled in release configuration
#endif
// Basic assertion - to be placed at upper level (where checks don't affect performance)
#define ASSERT(X,...) if(!(X)) exit(Crash(__VA_ARGS__));
#define ULL(X) ((unsigned long long)X) // Use %llu to print indices

// Memory allocations
template <typename T> T* GimmeMem(size_t N, const char* label=NULL); // Allocation wrapper
template <typename T> void FreeMem(T* &ptr); // Deallocation wrapper
```

Figure 2. Code fragments that illustrate index types, error handling and memory allocation definitions

There are explicit checks that ensure that the actual index value fits the array range. The runtime error handling is implemented in two configurations: safe mode and release. The safe mode configuration enables all the low-level checks in performance-critical areas (*SAFE_ASSERT* definition in Fig. 2). The release configuration only performs upper-level checks (*ASSERT* definition) that don't affect performance. If some assertion fails, the *Crash* function is called. It prints complete diagnostics information and aborts the execution. Further details will be given in Section 5.

An array of values for a set of mesh objects of a certain kind is represented as a template class wrapped around a plain pointer. This class, denoted as *tArray* (see Fig. 3), knows the size of the array and its name and provides explicit access check at runtime. Access to elements of a *tArray* object via *operator[]* is only allowed for an index of the same kind defined by the first template parameter. This prevents confusing kinds of sets at compilation time. Source code fragments with selected relevant methods are shown in Fig. 3.

```
template <tIdxKind I, typename T> class tArray { // Basic 1D array
protected:
    T *VV;       // Pointer to data vector
    size_t NN;   // Size
    string name; // Text label (for memory reports and error messages)
    inline void init_vec(string lbl=""){ NN=0; VV=NULL; name=lbl; } // Defaults
public:
    inline void Alloc(tIdx<I> n, string lbl=""){
        ASSERT(NN==0,"tArray::Alloc %s: already allocated", lbl.c_str());
        ASSERT(n>0 , "tArray::Alloc %s: wrong size %llu",   lbl.c_str(), ULL(n));
        VV=GimmeMem<T>(n.idx(), lbl.c_str()); NN=(size_t)n; name = lbl;
    }
    inline void Dealloc(){ if(NN>0 && VV) FreeMem(VV); init_vec(""); }

    tArray(string lbl=""){ init_vec(lbl); }
    tArray(tIdx<I> n, string lbl=""){ init_vec(); Alloc(n, lbl); }
   ~tArray(){ Dealloc(); }

    inline T &operator[](tIdx<I> i){
        SAFE_ASSERT(i.idx()>=0 && i.idx()<(tInt)NN, "tArray %s[%llu] out of size %llu",
                    name.c_str(), ULL(i.idx()), ULL(NN));
        return VV[i.idx()];
    }
//  ... whatever other necessary functions and operators
};
```

Figure 3. Code fragments of the 1D array template class that stores data for a set of mesh objects of a certain kind

2D arrays (in other words, block arrays) are represented by the *tBlockArray* class (see Fig. 4). It is assumed that the first index of the 2D array is the number of the block that corresponds to the mesh object in the set, and the second index is the position inside the block. Blocks store multiple values per mesh object, for instance, values of mesh functions defined in objects of a certain kind. This template class has two parameters, the kind of objects and the type of values. The underlying data storage is also a plain array. In the release configuration *operator[]* just returns the pointer to the requested block. In the safe mode configuration it returns an object of the *tBlock* class. This class is a wrapper around the pointer to the block that performs access correctness check inside the block.

Finally, we need block arrays with variable block size for adjacency data (mesh topology, portraits of sparse matrices, etc.) from Section 1. Such a container is represented by the *tVBlockArray* class (see Fig. 5). Its implementation is similar to block arrays, *operator[]* also returns a *tBlock* object. The difference is that there is a plain CSR-based (or whatever else suitable format) structure behind this wrapper.

In summary, the simplified set of containers consists of the *tArray* class for 1D arrays, the *tBlockArray* and *tVBlockArray* classes for 2D arrays with fixed and variable block sizes,

```
template <typename T> class tBlock { // A block of a block array (return type for operator[])
private:
  T *V; // Data pointer
  const char *name; // Pointer to parent object's name
  int M; // Block size (its not so big, 32-bit is ok)
  inline tBlock(){ M=0; V=NULL; } // Forbidden
public:
  inline tBlock(const tBlock<T> &b){ M=b.M; V=b.V; name=b.name; }
  inline tBlock(int m, T *v, const char *lbl=NULL){ M=m; V=v; name=lbl; }
  inline T& operator[](int i){
    SAFE_ASSERT(i>=0 && i<M, "tBlock %s[][%d] out of size %d", name, i, M);
    return V[i];
  } // ... other operators and functions ...
};

#ifdef SAFE_MODE
#define BLOCK(T) tBlock<T> // A block with safe mode access check
#define PTR2BLOCK(T,M,P,L)/*Pointer-to-block*/ tBlock<T>(M/*block size*/,P/*pointer*/,L/*label*/)
#else
#define BLOCK(X) X* // A plain pointer in release
#define PTR2BLOCK(T,M,P,L) P
#endif

template <tIdxKind I, typename T> class tBlockArray : public tArray<I,T>{
protected:
  tInt N; // Number of blocks
  int M;  // Block size
public:   // ...
  inline BLOCK(T) operator[](tIdx<I> i){
    SAFE_ASSERT((i>=0 && i<N),"tBlockArray %s[%llu] out of size %llu",
                tArray<I,T>::name.c_str(), ULL(i.idx()), ULL(N));
    return PTR2BLOCK(T, M, (tArray<I,T>::VV+(size_t)(i.idx()*M)), (tArray<I,T>::name.c_str()));
  } // ... other operators and functions ...
};
```

Figure 4. Code fragments that illustrate how block arrays work

```
template <tIdxKind I, typename T> class tVBlockArray : public tArray<I,T>{
protected:
   tInt N;   // Size - number of blocks (rows)
   tInt *IA; // CSR-like block (row) pointer offsets (of N+1 size)
public:      // ...
   inline int BlockSize(const tIdx<I> i)const{
      SAFE_ASSERT(i>=0&&i<N, "tVBlockArray::BlockSize %s[%llu] out of size %llu",
                  tArray<I,T>::name.c_str(), ULL(i.idx()), ULL(N));
      SAFE_ASSERT(IA[i.idx()+1]-IA[i.idx()]>=0 && IA[i.idx()]>=0,
                  "tVBlockArray::BlockSize %s: error in IA", tArray<I,T>::name.c_str());
      return (int)(IA[i.idx()+1]-IA[i.idx()]);
   }
   inline BLOCK(T) operator[](tIdx<I> i){
      SAFE_ASSERT(i>=0 && i<N, "tVBlockArray %s[%llu] out of size %llu",
                  tArray<I,T>::name.c_str(), ULL(i.idx()), ULL(N));
      return PTR2BLOCK(T, BlockSize(i), (tArray<I,T>::VV+IA[i.idx()]), (tArray<I,T>::name.c_str()));
   }
   // ... other operators and functions ...
 };
```

Figure 5. Code fragments that illustrate how block arrays with variable block size work

respectively. These template wrappers around plain pointers provide complete access correctness check that includes range check for all indices (both indices of 2D arrays) at runtime and compile-time check of correctness of the kind. Illustrative code fragments are shown in Fig. 6 (note the node ranges at the top of the figure, which will be explained in Section 4).

3. Some Complications of the Data Structure

In the previous sections, we described a simplified model with just a few kinds of mesh objects. There are still some problems that need to be addressed, in particular, the representation of the boundary surface of the mesh and the data sets associated with the cells.

```
    tIdx<IDX_N> nn; // Number of nodes in MPI extended subdomain
    tIdx<IDX_N> n_beg, n_end; // Range for extended subdomain (owned+halo nodes)
    // Subsets (vertex-centered case, mesh decomposition for the nodal graph)
    tIdx<IDX_N> n_owned_beg, n_owned_end; // Range for owned nodes (inner+interface)
    tIdx<IDX_N> n_inner_beg, n_inner_end; // Range for inner nodes
    tIdx<IDX_N> n_iface_beg, n_iface_end; // Range for interface nodes
    tIdx<IDX_N> n_halo_beg,  n_halo_end;  // Range for halo nodes
    // ...
    tIdx<IDX_E> en; // Number of elements in MPI subdomain
    tIdx<IDX_E> e_beg, e_end; // Range for elements
    // ...
    tBlockArray<IDX_N, double> BAN; // Some block array over nodes
    tVBlockArray<IDX_E, tIdx<IDX_N> > EN_topo; // Elements-Nodes mesh topology
    // ...
    void DoSomething1(tIdx<IDX_N> in, BLOCK(double) v);
    void DoSomething2(tIdx<IDX_N> in, tIdx<IDX_E> ie, BLOCK(double) v);
    // ...
  //for(tIdx<IDX_N> in=e_beg; in<n_end; ++in) // Compile-time error
  //for(tIdx<IDX_E> ie=n_beg; ie<n_end; ++ie) // Compile-time error
    for(tIdx<IDX_N> in=n_beg; in<n_end; ++in) // Loop over nodes
        DoSomething1(in, BAN[in]);
    // ...
    for(tIdx<IDX_E> ie=e_beg; ie<e_end; ++ie){  // Loop over elements
        tIdx<IDX_N> in;
        BLOCK(tIdx<IDX_N>) nodes = EN_topo[ie]; // The nodes of the element
  //    BLOCK(tIdx<IDX_N>) nodes = EN_topo[in]; // Compile-time error
  //    BLOCK(tIdx<IDX_E>) nodes = EN_topo[ie]; // Compile-time error
        for(int j=0; j<EN_topo.BlockSize(ie); ++j){
  //    for(int j=0; j<=EN_topo.BlockSize(ie); ++j){ // Runtime error at block access check
            in = nodes[j];
  //        in = ie; // Compile-time error
  //        DoSomething2(ie, in, BAN[in]);   // Compile-time error
            DoSomething2(in, ie, BAN[in]);
        }
    }
```

Figure 6. Code fragments that illustrate how basic containers for mesh data work

The objects of the mesh surface are represented with additional kinds, such as boundary nodes, boundary faces and elements. In this case, the data containers associated with the boundary cannot be accessed using indices of the volume mesh. On the other hand, a surface mesh is in fact a submesh of a volume mesh. Therefore, functions that work with the boundary must have access to the volume mesh data. For this purpose, we use index arrays that map the boundary numeration on the volume mesh numeration. Such an array of the *tArray* class stores for each boundary node its index in the volume mesh. Thus, access from the boundary submesh to the mesh data is organized with compile-time checking of correctness of the kind. Inverse mapping from the mesh to the boundary is implemented in a similar way. Inner nodes have negative values in this index array, which will result in an error if used in access to boundary data. The same approach is used for other submeshes, such as sliding interface zones. Note that we do not use associative containers, such as *std::map* and *std::unordered_map*, in order to preserve plain vectors (that we need for computing on GPU) and to avoid logarithmic cost of access (*std::map*).

Alternatively, the boundary surface could be represented with the same basic set of kinds. Then a boundary index cannot be distinguished from a volume mesh index. The correct access in this case would be ensured by using a specific structure of objects in which the boundary surface is separated from the volume mesh.

Finally, if the code deals with both vertex-centered and element-centered formulations, this makes a lot of problems. It is convenient to have a special kind for cells that is switching between nodes and elements, respectively. Since the choice of the formulation of the numerical scheme occurs at run time, we cannot control correctness of the kind at compilation time when accessing data in cells. This fact complicates the implementation, making it more nasty, however, correctness of the kind is still ensured. To access data associated with nodes and elements using an index associated with cells (and vice versa), we use the corresponding template cast operator (see Fig. 7). Converting a cell index is only allowed into a node index or an element index. Incorrect conversion into one of these kinds will be prevented at run time by the assertion that the numerical scheme is of the proper formulation. Conversion to any other kind will be prevented at linking time (since such a function is not defined).

```
template<tIdxKind I> class tIdx {
// ...
public:
    template<tIdxKind TT> tIdx(const tIdx<TT> &I); // Conversion of kinds
};

bool IsCC(); // Checks if the numerical scheme is cell-centered.
// Allowed index kind conversions
template<>template<> inline tIdx<IDX_N>::tIdx(const tIdx<IDX_C>& j){SAFE_ASSERT(!IsCC(),"Wrong C2N"); i=j.idx();}
template<>template<> inline tIdx<IDX_C>::tIdx(const tIdx<IDX_N>& j){SAFE_ASSERT(!IsCC(),"Wrong N2C"); i=j.idx();}
template<>template<> inline tIdx<IDX_E>::tIdx(const tIdx<IDX_C>& j){SAFE_ASSERT( IsCC(),"Wrong C2E"); i=j.idx();}
template<>template<> inline tIdx<IDX_C>::tIdx(const tIdx<IDX_E>& j){SAFE_ASSERT( IsCC(),"Wrong E2C"); i=j.idx();}
// ...
tArray<IDX_C, double> A; // Array over cells
tIdx<IDX_N> in;
tIdx<IDX_E> ie;
tIdx<IDX_S> is;
A[ie] = 0.0;   // Correct access in cell-centered case
//A[in] = 0.0; // Runtime error – conversion exists but the assertion fails
//A[is] = 0.0; // Link-time error – conversion is declared but the function doesn’t exist
```

Figure 7. Code fragments that illustrate how index conversion works

4. Parallel Computing

We use multilevel parallelization, which combines different parallel programming models: a message-passing model for cluster systems with distributed memory, a shared-memory model for multicore processors, and a stream processing paradigm for computing on accelerators.

The Message Passing Interface (MPI) is used on the first level. The mesh is decomposed into subdomains in order to distribute workload among supercomputer nodes. The cells (nodes or elements) are assigned to MPI processes. The part of the mesh that each MPI process works with consists of its own (local) cells and halo cells, which are cells from neighboring subdomains coupled by the scheme stencil with owned cells. The set of owned cells is into interface cells (that are coupled with halo cells) and inner cells. This separation of inner and interface cells is used for hiding communications behind computations by overlapping data exchanges needed for the interface with computations over the inner set.

Since MPI processes work with their local parts of the mesh, we need containers that provide mapping between local numbering of the MPI process and global numbering:

- the local-to-global (**L2G**) mapping array that stores for each cell its index in the whole global mesh;
- the global-to-local (**G2L**) array that stores the inverse mapping.

The former is needed to write simulation results, and the latter is needed to derive the local mesh data from the global mesh data stored on disk. Both L2G and G2L arrays are represented with *tArray* containers, and, of course, the global index has a different kind. The G2L array stores a negative index value for objects that do not belong to the MPI subdomain. This certainly leads to some memory overhead proportional to the global number of cells, but the access cost is constant. Again, as in the case of mapping for the boundary surface, we do not use associative containers (maps). In addition, it should be noted that the *std::unordered_map* containers may also consume a lot of memory for hash data.

For a mesh that is so big (billions of cells) that the G2L array cannot fit in memory, we can use multilevel partitioning and distributed storage. Firstly, the mesh is decomposed into small enough pieces stored in separate files. Then these pieces are fragmented further into the needed number of MPI subdomains. In this case, none of the MPI processes works with the entire mesh at all, and the G2L mapping is limited to a sufficiently small subdomain of the upper-level decomposition. It should also be noted that the cells are reordered several times at the initialization stage. Firstly, the cells are ordered by the owned/halo sets, so that the owned cells always have a smaller index that the halo cells. Then, the halo cells are ordered ascending ranks of their owners, and the owned cells are ordered by inner/interface sets (see Fig. 8).

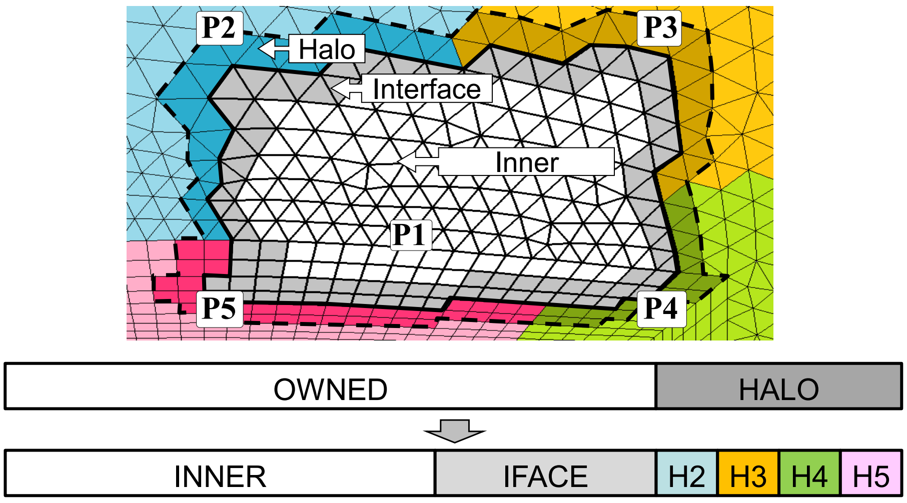


Figure 8. Reordering of cells in MPI subdomains

The OpenMP open standard is used for shared-memory parallel model on the second level. Instead of applying OpenMP loop parallelization directives, each MPI subdomain is further decomposed into second-level subdomains of OpenMP threads. Accordingly, the cells are reordered ascending their thread numbers. Then, each thread reorders its set of cells using Cuthill–McKee algorithm [3] in order to improve memory access pattern.

With this fixed decomposition, the sets of mesh objects associated with threads remain constant. This fact allows us to use the "first touch rule" in order to allocate data on NUMA system more efficiently. Each thread initializes its data in memory to help it be located in the nearest physical memory of the corresponding processor core. This would not make much sense in the case of loop parallelism, because loops for objects of different kind operate with data distributed differently among threads.

Finally, the OpenCL open standard is used for computing on accelerators of various architecture. Our containers are supplemented with OpenCL buffer handles for representing data on accelerators (devices). Since the containers are based on plain continuous 1D arrays, data can be transferred directly between the host and the devices without any conversions.

Regarding the representation of block arrays on GPUs, we use transposed access to block arrays in OpenCL kernels to improve memory access pattern (see [9]). The problem is that when blocks of a block array are associated with OpenCL threads, neighboring threads read their values with a non-unit stride (the distance is equal to the size of blocks), which is inefficient. In order to achieve coalescing of memory transactions, each local workgroup firstly reads its fragment of a block array into the shared local memory linearly (neighboring threads read neighboring values in memory, as if the fragment was transposed) . Then the threads of the local workgroup access this fragment normally (as a block array), but in the fast local memory. This approach gives us an efficient memory access pattern and preserves compatibility with the host-side data structure. Additionally, fragments of local workgroups in block arrays can be aligned (using padding) to the cache line size, so that the memory access is properly aligned.

Further information on parallelization technology can be found in [8, 9].

5. Runtime Diagnostics and Memory Allocation

We have upper-level and lower-level checks in the code. In simple terms, functions that have a loop over mesh objects of some kind are upper-level functions. Functions that process a single mesh object are lower-level functions. The former are computationally heavy, while the latter are light enough so that checks may produce some notable overhead. Therefore, error handling is implemented via two macro definitions, *ASSERT* and *SAFE_ASSERT*, for upper and lower levels, respectively (Fig. 2). Lower-level checks are deactivated in release configuration, so we don't waste time on high-frequency checks if everything goes fine. These checks are always enabled for quality assurance (QA) testing, as well as in the case if something goes wrong in order to quickly catch the problem.

Access errors can be easily localized, as our basic containers have names, and we know which array is it. Furthermore, we use an internal call stack tracker. In the case of an error, it reports the list of called functions or even particular code blocks that lead to the crash place. It is implemented as a class *tStackAgent* (see Fig. 9) that operates with a global singleton object, which, in turn, just stores some thread-private list of text labels. The constructor of this class adds a label of a function into the list, and the destructor removes it. These "agents" are manually placed at the beginning of functions or important code blocks via macro definitions *INFORM_KGB* and *INFORM_KGB_LOW* for upper and lower levels, respectively. Lower-level agents are disabled in the release configuration.

```
struct tStackAgent{
    tStackAgent(const char *Text, int Level=1); // Adds label to the list
   ~tStackAgent(); // Removes label from the list
};
#define INFORM_KGB(X)     tStackAgent KGB_agent(X/*name*/, 1/*level*/);
#ifdef SAFE_MODE
#define INFORM_KGB_LOW(X) tStackAgent KGB_agent(X/*name*/, 2/*level*/);
#else
#define INFORM_KGB_LOW(X) // disabled
#endif

void Function1_low_level(tIdx<IDX_E> ie){
    {
        INFORM_KGB_LOW("Function1_low_level block1");
        // ... some calculations ...
    }
}

void Function1_high_level(){
   INFORM_KGB("Function1_high_level");
   for(tIdx<IDX_E> ie=e_beg; ie<e_end; ++ie) // Loop over elements
       Function1_low_level(ie);
}
```

Figure 9. Code fragments of the built-in call stack diagnostics

If an assertion fails, the *Crash* function is called with an error message in *printf* format as input. It prints the message to the log file of the parallel process, and to the standard output and error streams. Additionally, it prints information about the actual call stack. If an error occurred at memory allocation, a full memory allocation report for all the arrays is also printed. Then the *Crash* function calls *MPI_Abort* that terminates the group of MPI processes. Note that the *Crash* call is wrapped with the *exit* function in order to inform the compiler about the exit point (for proper code analysis).

Now consider the memory allocation. For performance reasons, dynamic memory allocation is only allowed in upper-level functions. The direct usage of the *new* and *delete[]* operators is forbidden by our coding standard. Memory allocations are wrapped by the template function *GimmeMem* (Fig. 2). It takes the number of elements to be allocated and the text label of this allocation as input. It checks correctness and tries to allocate memory using the *new* operator, which is covered by the try-catch exception handing. If the allocation is successful, the pointer, its size and text label are added to some singleton object that stores the list of allocated pointers. The *Crash* function is called otherwise. This list is used for a full memory allocation report, if needed. It also gives us runtime protection from cyclic memory leaks (by checking the number of allocations). Deallocation is done by the *FreeMem* function, which calls the *delete[]* operator and removes the pointer from the list.

Finally, in order to save memory, the basic containers support the sharing (or aliasing) of data. For instance, if the portrait of a sparse matrix coincides with the mesh topology of some kind, it just shares this data.

6. Performance Evaluation

In terms of performance, we first need to make sure that our containers do not cause overhead in the release configuration compared to plain arrays. This will show the possibility of achieving maximum performance if the mesh data is properly reordered to minimize cache misses (which is beyond the scope of this work). Operations with block vectors with minimal arithmetic intensity were used for tests: assignment, elementwise addition, summation, etc. Different C++ compilers were used (Microsoft, GNU, Intel) on various computing equipment. Results for the release configuration demonstrated identical performance compared to plain arrays in all the tests. This means that compilers correctly substitute inline access functions with corresponding plain pointers.

Secondly, we need to check that the safe mode is not critically slower than the release configuration. The performance of the safe mode may seem unimportant at first glance (and usually nobody pays attention to it). However, when the safe mode is used in a very big simulation for debug purposes, reproducing the problem may consume notable amount of CPU hours just to initialize the simulation and perform several time steps. Therefore, the safe mode must not be by orders of magnitude slower. We tested sequential and parallel executions our supercomputer CFD code [8] on fine and coarse unstructured hybrid meshes (in a typical range from 10^5 to 10^6 cells per CPU socket) using explicit and implicit schemes on various computing equipment, including Lomonosov-2 supercomputer (14-core Intel Xeon Xeon E5 v3). The safe mode configuration appeared to be around 20% slower on average than the release configuration. It can be noted that in the case of an implicit scheme, which is mainly used in simulations, the overhead is smaller, around 10–15%. This is because of the solver for the sparse linear system with the Jacobi matrix, which must be solved at each iteration of the Newton process. The solver operates only with plain data and consumes nearly half of the overall computing time. Respectively, in case of an explicit scheme the overhead is bigger, around 30%. We can conclude that the observed overhead in the safe mode is insignificant for debug purposes in all the tests.

Finally, the overhead in compilation time has also been measured. It appeared to be around 5%, which is also negligible.

Conclusions

A set of containers with improved access safety has been proposed for storing mesh data in a CFD code. The presented containers for arrays, block arrays, and block arrays with variable block size can be associated with mesh objects of a certain kind. Such containers, which are in fact template wrappers around plain pointers to 1D data vectors, cause no overhead in the release configuration. At the same time, this approach provides throughout access control and extra protection from programming errors. The correctness of the kind of mesh objects is ensured at compilation time, while index range checking works at runtime.

Our QA procedure (about 200 short tests) for the safe mode configuration with low-level data access checks filters out programming errors much more efficiently. Those tricky errors that infiltrate through the QA procedure are captured and localized in the safe mode execution of a supercomputer simulation. To help bugs get caught, we use named arrays to know which data is being accessed incorrectly, a built-in memory allocation monitor to know distribution of memory costs and prevent cyclic memory leaks, an internal call stack tracker that helps to localize problems.

Using the safe mode configuration with improved-reliability containers helped us to notably reduce the costs of warfare against unstable "magic" bugs. It greatly simplified the debugging process and allowed us to improve the quality of our supercomputer simulation software.

Acknowledgements

The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University. The research is funded by the Russian Science Foundation, project 19-11-00299. The authors thankfully acknowledge these institutions.

This paper is distributed under the terms of the Creative Commons Attribution-Non Commercial 3.0 License which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is properly cited.

References

1. Alumbaugh, T.J., Jiao, X.: Compact Array-Based Mesh Data Structures. In: Hanks B.W. (eds) Proceedings of the 14th International Meshing Roundtable. Springer, Berlin, Heidelberg pp. 485–503 (2013), DOI: 10.1007/3-540-29090-7_29

2. Cantwell, C.D., Moxey, D., Comerford, A., Bolis, A., Rocco, G., Mengaldo, G., De Grazia, D., Yakovlev, S., Lombard, J.E., Ekelschot, D., Jordi, B., Xu, H., Mohamied, Y., Eskilsson, C., Nelson, B., Vos, P., Biotto, C., Kirby, R.M., Sherwin, S.J.: Nektar++: An open-source spectral/hp element framework. Computer physics communications 192, 205–219 (2015),

DOI: 10.1016/j.cpc.2015.02.008

3. Cuthill, E., McKee, J.: Reducing the Bandwidth of Sparse Symmetric Matrices. In: Proceedings of the 1969 24th National Conference. pp. 157–172. ACM '69, ACM, New York, NY, USA (1969), DOI: 10.1145/800195.805928

4. Danilov, A.A., Terekhov, K.M., Konshin, I.N., Vassilevski, Y.V.: INMOST Parallel Platform: Framework for Numerical Modeling. Supercomputing Frontiers and Innovations 2(4), 55–66 (2015), DOI: 10.14529/jsfi150404

5. Dyedov, V., Ray, N., Einstein, D., Jiao, X., Tautges, T.J.: AHF: array-based half-facet data structure for mixed-dimensional and non-manifold meshes. Engineering with Computers 31(3), 389–404 (2015), DOI: 10.1007/s00366-014-0378-6

6. Fogerty, S., Martineau, M., Garimella, R., Robey, R.: A comparative study of multi-material data structures for computational physics applications. Computers & Mathematics with Applications 78(2), 565–581 (2019), DOI: 10.1016/j.camwa.2018.06.010

7. Garimella, R., Perkins, W., Buksas, M., Berndt, M., Lipnikov, K., Coon, E., Moulton, J., Painter, S.: Mesh infrastructure for coupled multiprocess geophysical simulations. Procedia Engineering 82 (12 2014), DOI: 10.1016/j.proeng.2014.10.371

8. Gorobets, A.: Parallel Algorithm of the NOISEtte Code for CFD and CAA Simulations. Lobachevskii Journal of Mathematics 39(4), 524–532 (2018), DOI: 10.1134/S1995080218040078

9. Gorobets, A., Soukov, S., Bogdanov, P.: Multilevel parallelization for simulating turbulent flows on most kinds of hybrid supercomputers. Computers and Fluids 173, 171–177 (2018), DOI: 10.1016/j.compfluid.2018.03.011

10. Tautges, T.J.: MOAB-SD: Integrated structured and unstructured mesh representation. Engineering With Computers 20(3), 286–293 (2004), DOI: 10.1007/s00366-004-0296-0

11. Vazquez, M., Houzeaux, G., Koric, S., Artigues, A., Aguado-Sierra, J., Aris, R., Mira, D., Calmet, H., Cucchietti, F., Owen, H., Taha, A., Burness, E.D., Cela, J.M., Valero, M.: Alya: Multiphysics engineering simulation toward exascale. Journal of Computational Science 14, 15–27 (2016), DOI: 10.1016/j.jocs.2015.12.007

12. Weinbub, J., Rupp, K., Selberherr, S.: A Flexible Dynamic Data Structure for Scientific Computing. IAENG Transactions on Engineering Technologies. Lecture Notes in Electrical Engineering 229, 565–577 (2013), DOI: 10.1007/978-94-007-6190-2_43