The myriad (inter)faces of arrays

Ahmidas Development Team

October 26, 2008

By necessity, QCD oriented tools like Ahmidas will have to manage large blocks of structured memory. The efficiency of the code is determined to no small extent by the way the construction of, and access to, these blocks is arranged. Certain approaches, in particular those that are based on using coordinate systems directly, are particularly easy in scalar code with lots of low level access to the data. Unfortunately, these methods can quickly become unwieldy. They promote code with lots of non-contiguous memory access and can actually become fairly confusing when we move to parallel code. For that reason, it has been one of the design features of Ahmidas to implement most operations globally, such that special care can be given to implement things efficiently at the lowest levels and thereby unclutter the interface to be used in physically complicated parts of the code.

However, a certain amount of low level interfacing has to be provided for. The absence of such an interface was, in fact, found to be one of the major drawbacks of QDP++, which otherwise is built around the same principles (in fact, the current setup has QDP++ as its spiritual ancestor). If nothing else, well designed access to the underlying data structures makes for much less bug prone implementations of the basic operations. Additionally, one has to allow for the creative expansion of the scope of Ahmidas, perhaps introducing new operations that force access to data in innovative ways.

We have therefore tried to come up with a set of rules for data accessors. The idea being that in this way, we can implement uniform behaviour over different data structures, providing the user with as few surprises as possible. By providing a series of choices of well defined methods with their own pros and cons, we also try to empower the user. Certain breaches of data protection will be necessary for the most efficient ways of accessing data, but the user can choose to take full responsibility if he wishes to do so.

One consideration for the data interface that might not be immediately obvious is the risk of spurious copies being made. Within the Ahmidas framework, one can distinguish between low level data structures (such as SU3::Matrix) and their high level counterparts (with Core::Field being the canonical example). Their main difference lies in the underlying data management. Low level structures store their data in contiguous flat arrays and simply provide an interface to this data. Care is taken that they are sure to not store anything else than this raw data, which means that an array of, *e.g.* SU3::Matrix elements, by grace of the fact that structs store their data contiguously within C and C++, can be viewed as an array of it's lower lying elements. Concretely, QCD::Gauge* is equivalent to SU3::Matrix[4] is equivalent to std::complex<double>[36]. On the other hand, high level data structures are implemented using a copy-on-write reference-counting design pattern, which means that simply making a flat copy of such a structure is cheap, even if the amount of data accessed through the structure is in the order of hundreds of megabytes. Because of this, it is possible to write functions that return complex data structures by value, a very valuable property if one wants to create an interface that is as clean as possible. Several copies of a single data object will share their memory implicitly, up until the point that contained data is actually modified. A copy will then be forced if necessary (the private member **isolate** takes care of this), as is of course unavoidable if we in fact mean to store two independent sets of data.

The importance of all this for indexing lies in the final step of this scheme: the automatic copying of data structures if and when this is necessary. The only way to signal the compiler that a modification is upcoming, is through the specific accessor used. Of course, once we declare a data object const, this problem is solved. As a consequence, indexing on a const data structure, with whatever operator, calls an overloaded accessor that will never force a copy of the structure. There may, however, be cases in which we cannot declare an object constant, but where we would still like to be able to signal the compiler that we in fact do not intend (or, more precisely, shouldn't be allowed to) modify the data. For these cases, special accessors will be supplied as partners to the ones presented here. These should be employed to the end users discretion.

1 Index operators

In this section, we describe the different forms in which one can gain access to data components by use of a numerical reference, something we

refer to here as indexing. These operators take the form of a specific series of member functions, the archetype of which is the indexing operator of C++. The returned type can sometimes be slightly ambiguous, but we try to make sure it returns what one would normally (and unambiguously) consider a data element of whatever the data structure is. When it comes to containers such as Core::Field, the straightforward return type is the element contained within. For some more specialized containers, such as Core::Tensor, it is not completely well defined what the contained element is. In those cases, we choose to move down to the first well defined element for the generic indexing operators (such as physical indices), while some specialized operators using a coordinate representations can be used for alternative access modes. In practice, the latter are used far more rarely, so we find that the interface is cluttered least this way.

1.1 Physical indices

A natural approach to the indexing issue is using what we have dubbed the physical index, which is implemented (as a de facto default choice) with the overloaded **operator[]**. The term physical refers to the fact that this index automatically corrects for any internal offsets in the code. Though this of course means that memory access is not always contiguous and that some recalculation needs to be done in each step, it provides a way of using indices to run over two or more fields at the same time, where one is guaranteed that a particular index refers to a specific point on the 'physical' lattice. Exactly which point this is, is implementation dependent. On a scalar build of Ahmidas, there will in general be T points in a correlator, for example, but on an MPI implementation this number will be divided by the grid dimension in the time direction. To allow for an architecture independent way of running over all components of a data structure, all data structures provide the **size** member function. As described in the introduction, this operator has a counterpart leaving the data structure constant, designated **constPhysicalIndex**.

1.2 Memory indices

Though physical indices are easy to use, the added overhead of removing offsets may be unnecessary. For example, when reunitarizing all SU(3) matrices in a gauge field, we don't care about the particular order. For these cases, it is advantageous to have access to contiguous memory blocks and it is provided through the **memoryIndex** function. Obviously, the number

of elements is no different than for the case of physical indices, so again the **size** member function can be used. As described in the introduction, this operator has a counterpart leaving the data structure constant, designated **constMemoryIndex**.

1.3 Coordinate indices

With coordinate indices, we arrive at the first indexing operator that takes more than a single numerical value. Internally, coordinate indexing will perform a translation to a physical index and its performance should be comparable, but slightly slower. There are, however certain use cases where using coordinates is a very convenient alternative to using physical indices. Two obvious cases would be the placement of a point source on a lattice and the (internally implemented) contraction of two QCD::Tensor objects (that are represented as 12-by-12 matrices in Ahmidas). Note that Ahmidas provides an abstract layer on top of parallelization and that as such the coordinates are all global in nature, meaning that they run over a lattice of size $L^3 \times T$. To retrieve an element from the lattice given a specific set of coordinates, the accessor **at** is provided. As described in the introduction, this operator has a counterpart leaving the data structure constant, designated **constAt**.