Jayesh Badwaik March 4, 2019

Some Observations on Implementation of P1385

I have tried to develop a library following P1385. It is here. Here are my following observations regarding my experiences with the implementation.

1. We must separate algorithms from data structures in a philosophy similar to how STL has done with algorithms and containers.

2. This is somewhat a bike shedding topic. 'Engine' is more associated with something that computes rather than something that stores. Hence, we should name our data types as `storage` rather than `engine`. If you see my code, you can see that I am using the name engine as a function object to properly dispatch computations for basic operations.

3. Element promotion is tricky. We cannot do element promotion. Only the implementer of the elements can do element promotion. However, the element promotion section contains a non-related text. Probably the section heading should be changed.

4. The real text of element promotion section deals with verification of element being a field type. Neither integers nor floating points are field types. Hence, it is necessary to come up with a completely new definition from ground up which would allow us to to define what is a good field type for numerics. The paper uses a `is_matrix_element` as the name. I prefer a '`is_float_field`' as a name in my library but that is bike shedding and can be changed quickly. It is not possible to verify whether or not a type satisfies the required of being a field type by writing code for it. So, instead, we have two options
   - Treat all types which have a correct signature for 'operator+', 'operator-', 'operator*' and 'operator/' as field types.
   - Manually specify which types are field types.

5. Note that some of the algorithms WILL not work with integer types since the algorithms require $a \cdot a^{-1} = 1$ which an integer does not satisfy. So, do we want to have integers as float field types? This meshes with the previous question.

6. Minutes show that row and column vector were suggested to be merged. I would like a confirmation of that. I personally think that's a good decision.

7. **Vector of Dimension 1** should be synonymous with scalar type. This is important in generic code. How should this be handled? (I have handled it before by specify special operations for `vector<1>`. Here for example.

8. Types of parallelism/heterogeneity to deal with.
   - Parallel Algorithms
   - Parallel Data Structures

9. Before going there, lets recap one of the objectives. The objective is to be able to write

```
c = a+b;
```

where `a` and `b` are any matrices/vectors and the library will choose the correct executor to do the work.

10. Parallel/Heterogeneous Data Structure question is easier. The algorithm parallelism model can directly follow from the data structure model.

    For example, if the vector is stored on a GPU, then the sane choice is to carry out a full fledged GPU parallelism. Similarly, if a vector is a partitioned vector stored over multiple nodes in a cluster, each core has exclusive ownership of a part of matrix, and then the algorithm is straight forward.

11. For serial data structures, the question is a little tricky. Do we want it to be possible to use parallel executors for a data structure stored on CPU memory? If yes, who makes this decision?

    Ideally, the user would make the decision. But the user is only expected to specify operations in the manner of `a+b`. How do they specify the executor model?

    Do we want to further support `add(execution_policy,a,b)` type of functions for such special uses? And then we expect the user to call them when required?

    How does this fare in generic code? This is not a big problem, but a clear line of action is important.

12. Minutes show that there was a plan of abandoning arbitrary indexing in data structures. I would like a confirmation on that. A related note to make is that, if arbitrary indexes are supported, then signed sizes are more natural and (efficient?) in the code. However, I just spent 10 minutes looking for a bug because I was trying to use `std::size_t` instead of a signed type for my size variable.

13. If we agree to separate data structures and algorithms, then another question arises. Given an ordered set of known types, an appropriate trait will already exist in the library to determine the result type of the operations. However, if a user type is used, then there are two options to choose:

    - Refuse to compile unless the user specializes the type traits.
    - Choose a minimal type.

    I like the first option.

14. In all the above discussion, it will be noticed that it is important to allow users to specialize most type traits to choose the sane options (sane in my opinion). This has not been the position of the standard in general. Would an exception in this scenario be palatable to the standard? Considering that la will have its own namespace, this should not be an issue.

15. One can see from my current implementation that I am not caring about expression templates right now. However, one can also see that the implementation provides opportunity to implement expression templates in a good manner. For example, one can find out if a temporary was captured in the arguments, in which case, instead of an expression template, an actual value can be returned. (I think this can be achieved although the details are a little unclear at this point.)