A Generic Multi-Dimensional Run-Time Data Structure for High-Performance Scientific Computing

Josef Weinbub, *Member, IAENG,* Karl Rupp, and Siegfried Selberherr

***Abstract*—We present an approach for a generic, multi-dimensional run-time data structure suitable for high-performance scientific computing in C++. Our approach for associating meta-information with the data structure as well as different underlying datatypes is depicted. The high-performance, multi-dimensional data access is realized by utilizing a heterogenous compile-time container generation function. The generalized data structure implementation is discussed as well as performance results are shown with respect to reference implementations. We show that our approach is not only highly flexible but also offers high-performance data access by simultaneously relying on a small code base.**

***Index Terms*—multi-dimensional, data structure, Generic Programming, Meta Programming, C++**

I. Introduction

The plethora of applications in the field of scientific computing introduces different requirements for data structures. For example, a two-dimensional matrix is used for storing a linear system of equations [1]. On the contrary, a simulation may require a set of second-order tensors to describe, for instance, the stresses in the field of continuum mechanics [2]. Typically, the dimensionality is known during compile-time, thus the data structure can be optimized by the compiler [3]. However, for a unified data structure interface, a run-time solution is required. In this context, unified relates to a single datatype used to reflect data structures of arbitrary dimensionality. Such an application scenario arises, for example, at plugin interfaces within a software component framework. Each plugin provides and receives data, though the dimensionality is not known in advance. Therefore, an approach is required to support multiple dimensions during run-time.

An additional challenge arises with respect to the underlying datatype of the data structure elements. C++, as a statically typed programming language, enables type checks during compilation. This not only allows for the compiler to optimize code, but also to detect errors at the earliest possible stage of the development. However, such a system imposes restrictions in regard to the run-time handling of datatypes. For example, a floating-point number of type `double` can only hold a double-precision value, but not a `string` object.

Manuscript received March 5, 2012; revised March 29, 2012. This work has been supported by the European Research Council through the grant #247056 MOSILSPIN. K.Rupp acknowledges support by the Austrian Science Fund (FWF), grant P23598.

J.Weinbub, K.Rupp, and S.Selberherr are with the Institute for Microelectronics, Technische Universität Wien, Vienna, Austria.
e-mail: {weinbub,rupp,selberherr}@iue.tuwien.ac.at.
K.Rupp is additionally with the Institute for Analysis and Scientific Computing, Technische Universität Wien, Vienna, Austria.

This limitation is in principal desired, but it introduces challenges, when a generic data structure has to be implemented, where the type is not known in advance. Related to the previously introduced example of a plugin framework, the data structure at the plugin interface cannot only vary in its dimension but also in its type.

The field of scientific computing not only processes sets of values, but typically also sets of quantities. A quantity is referred to as a value which is associated with a unit. Supporting or even enforcing units is a vital part for ensuring the correctness of scientific simulations [4]. However, units may not be the only additional meta-information. For example, data values can be related to measurements carried out at a specific temperature. Overall, the need for a flexible property system arises, which should not only reside in the run-time domain, but also be orthogonal to the data structure. In this context, orthogonality refers to exchanging the data structure without influencing the attached meta-information. Such an approach is highly versatile, as it introduces exchangeability.

The continually growing demand for increased simulation performance introduces the need to parallelize simulation tools. Ideally, the individual computations should scale beyond a multi-core processor, namely to a distributed computing environment. Typically, the Message Passing Interface (MPI) is utilized for communication within a distributed environment. The data structure should support seemless integration into such an MPI based environment, to ease the integration process. Therefore a serialization approach for the data structure should be available, which allows out-of-the-box transmission by an MPI communication channel.

We introduce an approach for a flexible data structure in C++, which handles multiple dimensions, run-time generation, and supports different underlying datatypes. Additionally, we support direct transmission capabilities over MPI and an orthogonal and flexible coupling of meta-information with the data structure. We achieve this by utilizing modern programming techniques, in particular generic [5] and meta-programming [6], and the Boost Libraries [7]. We show that our approach does not only provide a high degree of flexibility, but also offers high-performance data access. Additionally, due to the heavy utilization of libraries in conjunction with the application of modern programming techniques, the required code base can be kept to a minimum. This fact significantly improves the maintainability of our implementation.

This work is organized as follows: Section II puts the work into context. Section III introduces our approach in detail and Section IV depicts performance results.

II. Related Work

A flexible run-time data structure for multi-dimensional problems is provided by Marray [8]. Marray is a C++ header-only library and publicly available under the MIT License. The library provides not only the generation of multi-dimensional arrays during run-time, but also views on sub-spaces of the generated data structures. A C++98 compliant implementation is available as well as a C++11 version, which utilizes, for example, the variadic template [9] mechanism to provide dimension independent access to the data structure.

Several multi-dimensional array libraries are available for the case of fixed dimensions during run-time. For example, the Boost MultiArray Library [10] and the Blitz++ Library [11] provide the generation of multi-dimensional arrays during compile-time. Additionally, views are provided to access a specific subset of the generated data structures.

III. Our Approach

Our approach focuses on several key-aspects, being:

1) A Polymorphic Datatype
2) Data Structure Generalization
3) Attaching Meta-Information
4) Serialization

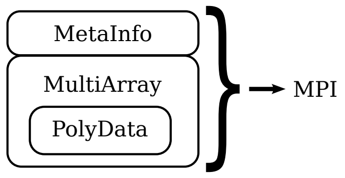

Fig. 1: Our approach is based on a polymorphic datatype which is used by the multi-dimensional array data structure. Meta-information is orthogonally coupled with the data structure. The overall approach is serializable, thus any object can be transmitted over an MPI communication channel.

Figure 1 depicts an overview of our approach. First, a polymorphic datatype[1] supporting different datatypes during run-time is introduced. Second, the polymorphic entries are embedded in a multi-dimensional run-time array data structure. Third, run-time meta-information is attached to the data structure. Fourth, our implementation is serialized to enable convenient transfer by the MPI. Throughout this section we discuss each of these aspects in detail and provide our implementation approach.

A. A Polymorphic Datatype

One of the core aspects of our approach is the ability to support different datatypes during run-time. The challenge is to provide one datatype which can in fact hold several different types. This is a peculiar task for statically typed languages, like C++, as the type system only allows to assign objects of the same or convertible type. If the types are not the same, cast operations have to be performed. However, applying casts can result in information loss, for example, when a datatype of higher precision, like `double`, is transformed to a datatype with lower precision, like `float`.

[1]Polymorphy denotes the ability to represent different datatypes

We utilize the Boost Variant Library (BVL) [12] for supporting different datatypes during run-time. Informally, a BVL datatype can be seen as an `enum` for datatypes. A set of possible, supported datatypes has to be provided during compile-time. During run-time, the instantiated BVL object can be associated with any of these datatypes.

We identify four different categories of datatypes, which are listed in the following:

- signed integer
- unsigned integer
- floating-point
- string

A meta-function for the generation of the polymorphic datatype based on the introduced four categories is provided. A meta-function is a class or a class template which provides a nested `type` typedef [13]. In the following, this mechanism is introduced in detail. First, the set of supported types is generated by utilizing an associative heterogeneous container provided by the Boost Fusion Library (BFL) [14], as depicted in the following.

```
1 typedef make_map<
2    Signed, UnSigned,     Float,  String,
3    int,    unsigned int, double, string
4  >::type                         Types;
```

The `make_map` meta-function is utilized to generate an associative BFL container (Lines 1-4). Note that the tags in Line 2 represent the individual categories, and the datatypes in Line 3 relate to the corresponding datatypes. Tags are typically implemented by so-called empty structures, for example, `struct Signed{};`. Generally, in our approach the datatypes can be set non-intrusively, meaning that the underlying datatypes can be exchanged, thus significantly improving the applicability and extendability of our aproach. For example, instead of the `double` floating-point datatype, a multi-precision datatype provided by the GNU Multiple Precision Arithmetic Library (GMP) [15] can be used, which would significantly improve the accuracy of subsequent floating-point operations.

In the following, the associative `Types` container is converted into a Boost Metaprogramming Library (MPL) [13] vector container by the `generate_typeset` meta-function.

```
typedef generate_typeset<Types>::type TypeSet;
```

This step is necessary, as the subsequent step of utilizing the BVL is eased, when the supported datatypes are available as an MPL sequence. A default implementation is available, which allows convenient generation of this typeset and only relies on built-in datatypes as shown in the following.

```
typedef generate_typeset<>::type      TypeSet;
```

The typeset is then used to generate the actual polymorphic datatype based on the BVL. Again a meta-function is used to generate the polymorphic datatype as depicted in the following.

```
typedef generate_polyvalue<
  TypeSet>::type                      PolyValue;
```

Internally, the BVL `make_variant_over` meta-function is utilized to generate the actual polymorphic datatype.

Finally, due to the BVL, it is possible to provide a generic way to support different datatypes during run-time:

```
PolyValue signed_integer  =
  value_at_key<Types, Signed>::type(4);
PolyValue floating_point =
  value_at_key<Types, Float>::type(4.0);
```

A type-safe approach for a signed integer and a floating-point datatype instantiation is implemented by using the BFL meta-function for key-based element access (`value_at_key`). Type safety is accomplished in this case, by accessing the actual type in the previously provided `Types` container.

B. Data Structure Generalization

Based on the previously introduced polymorphic datatype the actual array data structure can be implemented. The implementation has two goals: First, multiple dimensionality should be supported during run-time. Second, the data access should be as fast as possible. In this work we do not focus on advanced functionality, as, for example, provided by the Marray library. Instead, we aim for a straightforward data structure, coordinate-based access, and a high-performance implementation.

The following code snippet outlines the creation of a two-dimensional data structure, where the first and second dimension holds three and four elements, respectively.

```
1 typedef MultiArray<PolyValue>    MultiArrayT;
2 MultiArrayT::dimensions_type     dim;
3 dim.push_back(3);
4 dim.push_back(4);
5 MultiArrayT                      multiarray(dim);
```

The MultiArray implementation can be configured to hold arbitrary value types. In this case, the previously introduced polymorphic value type based on the BVL is used (Line 1). The dimensions are formulated by utilizing a Standard Template Library (STL) `vector` container [16], where the type is accessed by the member-type `dimensions_type` (Line 2). Each element of the dimensions container holds the number of elements of the respective dimension (Lines 3-4). The number of dimensions is therefore inherently provided by the size of the container. A MultiArray object is instantiated with the dimension configuration (Line 5). Internally, an STL `vector` container, which represents a linear memory block, is used. Data Structures of arbitrary dimensionality are mapped on this linear container, as depicted in Figure 2 for the two-dimensional case. The individual columns of the respective domains are stored consecutively. This approach minimizes the allocation time, as only one memory allocation step is necessary. However, a linear storage approach requires index handling to map the coordinate index tuple on the corresponding position within the linear data structure. This is the performance critical part, as the data access implementation is likely to be called on a regular basis.

The central challenge of providing coordinate index access is the handling of data structures of arbitrary dimension, as the number of access-indices correspond to the number of dimensions. Typically, the elements of a two dimensional array are accessed in coordinates, like `array(i,j)`. From the software development point of view, the challenge is to implement an access mechanism which is both high-performing and can be used for arbitrary dimensionality.

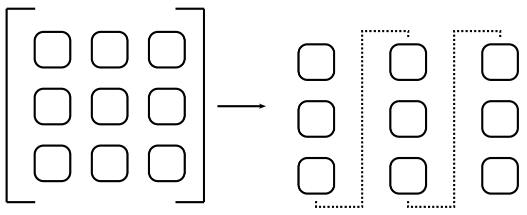

Fig. 2: A two-dimensional array is mapped on our internal, one-dimensional data structure.

Several approaches for the access implementation and the related index computation have been investigated. One approach is based on so-called variadic functions provided by the C programming language [17]. The primary drawback of this approach is the fact that the number of indices has to be provided explicitly. In addition, this approach is not type-safe, as it is based on macros. Another approach is based on utilizing an STL `vector` for the index container. The number of indices can easily vary during run-time. However, this approach suffers due to run-time overhead for the creation and the traversal of the index-vector for each data access. In the end, a BFL vector sequence is utilized, which offers superior performance due to the fact that the sequence is a compile-time container. The run-time generation is performed by a generation function provided by the BFL, like depicted in the following.

```
multiarray(make_vector(2,3)) = Numeric(3.5);
```

Note that the BFL generation function (`make_vector`) is utilized to generate the compile-time index container in-place. This can be considered a drawback with respect to usability, as it requires additional coding. However, convenience specializations can be implemented to hide the vector generation step from the user. Due to restrictions of the C++98/03 standard, these specializations can only be provided for a finite set of dimensions, thus such an approach cannot be considered truly multi-dimensional. Internally, the BFL `make_vector` function relies on a macro, for the generation of arbitrary dimensional compile-time data structures. However, we consider this to be an excellent compromise, at least until the C++11 standard is broadly available. This standard introduces the so-called variadic template mechanism, which is also applied by Marray in its C++11 extension. Our investigations revealed, that with variadic templates the same performance as with our current approach can be achieved but simultaneously the required access interface (`multiarray(2,3)`) for arbitrary dimensions can be realized.

Our high-performance index computation is implemented based on the BFL algorithms which allow partial compile-time computation, as depicted in the following.

```
1 template<typename IndexSequ>
2 Element& operator()(IndexSequ const& indices) {
3  return container[
4   accumulate(pop_front(indices),at_c<0>(indices),
5    make_index<dimensions_type>(dimensions)
6   ) ]; }
```

Lines 4-5 compute the actual index, which is then used to access the element in the linear container in Line 3. The index computation is based on various BFL mechanisms, like, `accumulate`.

For example, for a two-dimensional problem, the index is evaluated as follows: $i = I_0 + I_1 \cdot D_0$, where I_0 and I_1 refer to the first and second index, respectively. D_0 relates to the number of elements in the first dimension. This procedure can be extended to arbitrary dimensionality.

The introduced data structure generalization approach can also be applied for the tensor-valued elements. Therefore, a two-level hierarchy of the proposed MultiArray data structure supports tensor datasets of arbitrary dimension and varying datatypes.

C. Attaching Meta-Information

Scientific computations do not merely process values, but physical quantities. This is a subtle difference, as the former indicate simple values, like, a `double`, but the latter, associates the respective value with a unit, promoting the value to a physical quantity. This is rather important, as scientific computations should not just imply a unit system, they should enforce it to eliminate unit-related errors [4]. Keeping in mind that units might not be the only additional property which can be associated with a dataset, a flexible approach is required to associate additional meta-information with the data structure.

Another important aspect, however, is to ensure extendability and exchangeability. As such, the approach has to support the exchange of the data structure as well as the meta-information package. For example, it should be possible to exchange our data structure with the Marray implementation [8], without changing the associated meta-information package. Such an approach is considered orthogonal, as the exchange of one part does not influence the behavior of another part. Obviously, the implementation of the meta-information package has to be non-intrusive with respect to the data structure. More concretely, the package should not be placed inside the data structure class, but externally associated with it.

We implement an approach for storing arbitrary meta-information during run-time, which is straightforwardly based on the STL `map` container.

```
typedef map<string, string>     MetaInformation;
MetaInformation  minfo;
minfo["unit"] = "kg";
```

This approach is very flexible, as arbitrary properties can be added. Most importantly, though, the implementation effort is kept to a minimum, as already available functionality is utilized.

Finally, the data structure and the meta-information is coupled by the associative container of the BFL, which has already been introduced in Section III-A.

```
typedef make_map<
  data,         metainf,
  MultiArrayT, MetaInformation
>   QuantityDataset;
```

Note the orthogonal and extendable association of the data with additional properties. Orthogonality can be identified when, for example, exchanging `MultiArrayT` with the corresponding Marray datatype, which neither has an impact on the associated meta-information package `MetaInformation` nor on the overall handling of the `QuantityDataset`.

Extendability refers to the fact, that, by adding additional tags, further data can be associated with the dataset.

If the availability of a unit should be enforced, then the unit information should be moved from the `minfo` container one level up to the `QuantityDataset`. By utilizing a new tag and a string value, a `QuantityDataset` expects the unit information, like depicted in the following.

```
typedef make_map<
  data,         metainf,         unit,
  MultiArrayT, MetaInformation, string
>   QuantityDataset;
QuantityDataset quantity_dataset = make_map<
  data,metainf,unit>(multiarray,minfo,"kg");
```

This also outlines the flexibility of our approach, as different setups of the `QuantityDataset` can be enforced.

Our approach presumes that the unit applies to the complete dataset. In case heterogeneous units should be supported, an additional layer has to be introduced assigning a unit to a specific value.

D. Serialization

Serialization refers to the process of storing and retrieving the elements of a data structure. Typically, input/output mechanisms utilize serialization processes, as, for example, a matrix is written to a file. This is usually performed by implementing dedicated file writer functions. However, a major disadvantage is, that for each new file format a new writer has to be implemented. One approach to ease the burden of serialization, is to introduce an additional layer, which provides a common ground between the data structure and the target storage format. Thus, it is possible to implement the serialization mechanism for a data structure once, and then access the already available functionality based on the additional layer. However, serialization cannot only be used for file input/output processes, but also for MPI communication [18]. For this purpose we utilize the Boost Serialization Library (BSL) [19], which provides a serialization facility for arbitrary data structures. Based on our previously introduced quantity dataset (Section III-C), serialization extensions have been implemented. In the following, Process 0 transmits an available quantity dataset to Process 1.

```
if (world.rank() == 0) {
  for_each(quantity_dataset, send(comm, 1));
}
```

Process 1 receives the quantity dataset from Process 0:

```
if (world.rank() == 1) {
  QuantityDataset  quantity_dataset;
  for_each(quantity_dataset, recv(comm, 0));
}
```

Note that the unary auxiliary functor `send`/`recv` gets an element of the quantity dataset, being a BFL pair data structure, and sends/receives the data element of the respective pair. The BFL `for_each` algorithm is utilized to traverse the elements of the quantity dataset. Finally note, that additional convenience levels can be implemented to further wrap code away from the user. For example, a generic serialization implementation can be provided, which is capable of handling arbitrary BFL data structures.

IV. Performance

This section presents performance results for our data structure, especially our BFL based index computation approach. The tests have been carried out on an AMD Phenom II X4 965 with 8 GB of memory running a 64-Bit Linux distribution. The GNU GCC compiler in version 4.4.5 is used with the flags `-O2 -DNDEBUG`. Benchmarks are averaged over five runs to reduce noise. The element access performance for various problem sizes and different array dimensions is depicted, based on storing `double` values. The reference implementation is based on a hierarchy of STL vectors, as no index computation is required for the element access procedure. Additionally, we compare our approach with the already mentioned, publicly available Marray library [8]. Furthermore, we investigate the influence of optimal and non-optimal traversal, identified with OPT and NOPT, respectively. In the optimal case, the element access is as sequential as possible, meaning that the elements are accessed in the same consecutive manner as they are stored in the memory. Sequential access is favored by the so-called prefetching mechanism [20]. We investigate the non-optimal case, by exchanging the traversal loops for the two- and three-dimensional problems.

Figure 3 depicts the results for a one-dimensional array. Our approach is equally fast as the reference implementation, and takes around 0.15 seconds for writing data on all 10^8 elements. The Marray implementation is about a factor of 2.9 slower. The two-dimensional results are depicted in Figure 4. For 10^8 elements and the optimal traversal case our approach is again equally fast as the reference implementation, whereas Marray is a factor of 7 slower. In the non-optimal case, all implementations are significantly slower, and take approximately equally long (around 3.3 seconds for 10^8 elements). Figure 5 shows the results for a three-dimensional problem. Our optimal traversal implementation is a factor of 1.5 faster than the reference. For 10^8 elements Marray is a factor of 9.9 slower than our approach. As expected, the non-optimized traversal implementations are significantly slower, for instance, our optimal traversal approach is a factor of 48 faster than the non-optimized one.

Our implementation takes for all presented dimensions approximately equally long, which is not only due to the utilization of a linear storage but also due to the compile-time based index evaluation algorithm. This fact underlines the applicability for high-dimensional data storage applications.

V. Conclusion

We have introduced a flexible, multi-dimensional run-time data structure. Our approach offers high extendability, and can be applied in MPI based computing environments. The presented performance results depict that our access mechanism offers excellent performance for different dimensions and problem-sizes. The drawback of additional coding at the user-level access code will be rendered obsolete with the availability of variadic templates provided by the C++11 standard. Finally, our approach offers a small code base, as only around 100 code lines are required to implement the introduced functionality.

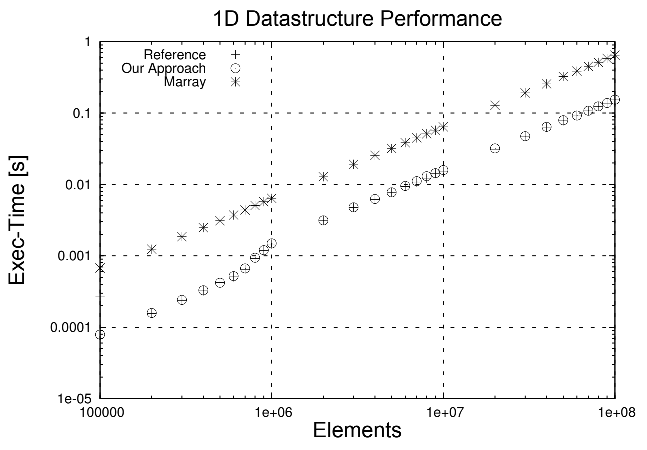

Fig. 3: A one-dimensional array structure is benchmarked. Our approach is equally fast than the reference implementation, whereas Marray is a factor of 2.9 slower.

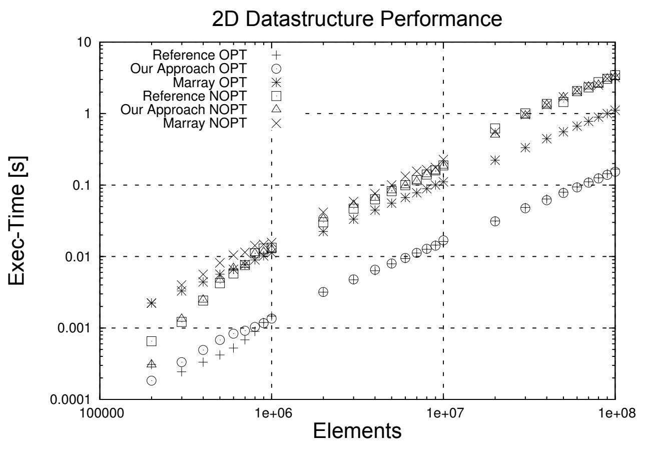

Fig. 4: A two-dimensional array structure is benchmarked. OPT and NOPT refers to optimal and non-optimal traversal, respectively. For 10^8 elements, our approach and Marray is a factor of 1.6 and 6.8, respectively, slower than the reference implementation. All approaches are equally slower in the non-optimized case, namely around 3.3 seconds for 10^8 elements.

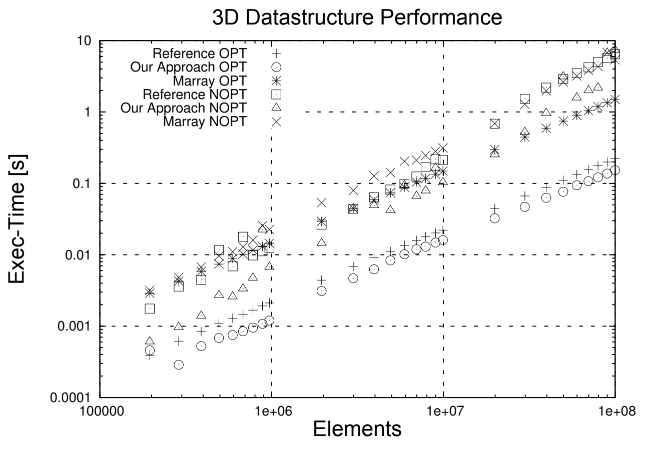

Fig. 5: A three-dimensional array structure is benchmarked. OPT and NOPT refers to optimal and non-optimal traversal, respectively. For 10^8 elements, our approach and Marray is a factor of 1.4 and 6.5, respectively, slower than the reference implementation. Non-optimal traversal significantly reduces the performance for all implementations.

References

[1] C. D. Meyer, *Matrix Analysis and Applied Linear Algebra*. SIAM, 2001.
[2] S. Nemat-Nasser, *Plasticity*. Cambridge University Press, 2004.
[3] A. Alexandrescu, *Modern C++ Design*. Addison-Wesley Professional, 2001.
[4] B. Stroustrup, "Software Development for Infrastructure," *Computer*, vol. 45, 2012.
[5] G. D. Reis *et al.*, "What is Generic Programming?" in *Proc. LCSD*, 2005.
[6] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming*. Addison-Wesley Professional, 2004.
[7] "The Boost C++ Libraries," http://www.boost.org/.
[8] B. Andres, U. Köthe, T. Kröger, and F. A. Hamprecht, "Runtime-Flexible Multi-dimensional Arrays and Views for C++98 and C++0x," *ArXiv e-prints, Technical Report*, 2010, http://www.andres.sc/marray.html.
[9] D. Gregor, J. Järvi, J. Maurer, and J. Merrill, "Proposed Wording for Variadic Templates," ANSI/ISO C++ Standard Committee, Tech. Rep. N2152=07-0012, 2007.
[10] R. Garcia and A. Lumsdaine, "MultiArray: A C++ Library for Generic Programming with Arrays," *Software: Practice and Experience*, vol. 35, no. 2, 2005.
[11] T. L. Veldhuizen, "Arrays in Blitz++," in *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE)*, 1998.
[12] "The Boost Variant Library," http://www.boost.org/libs/variant/.
[13] "The Boost Metaprogramming Library," http://www.boost.org/libs/mpl/.
[14] "The Boost Fusion Library," http://www.boost.org/libs/fusion/.
[15] *GNU Multiple Precision Arithmetic Library (GMP)*, http://gmplib.org/.
[16] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley, 2000.
[17] M. Banahan, D. Brady, and M. Doran, *The C Book*. Addison Wesley, 1991.
[18] "The Boost MPI Library," http://www.boost.org/libs/mpi/.
[19] "The Boost Serialization Library," http://www.boost.org/libs/serialization/.
[20] U. Drepper, "What Every Programmer Should Know About Memory," *Linux Weekly News*, 2007.