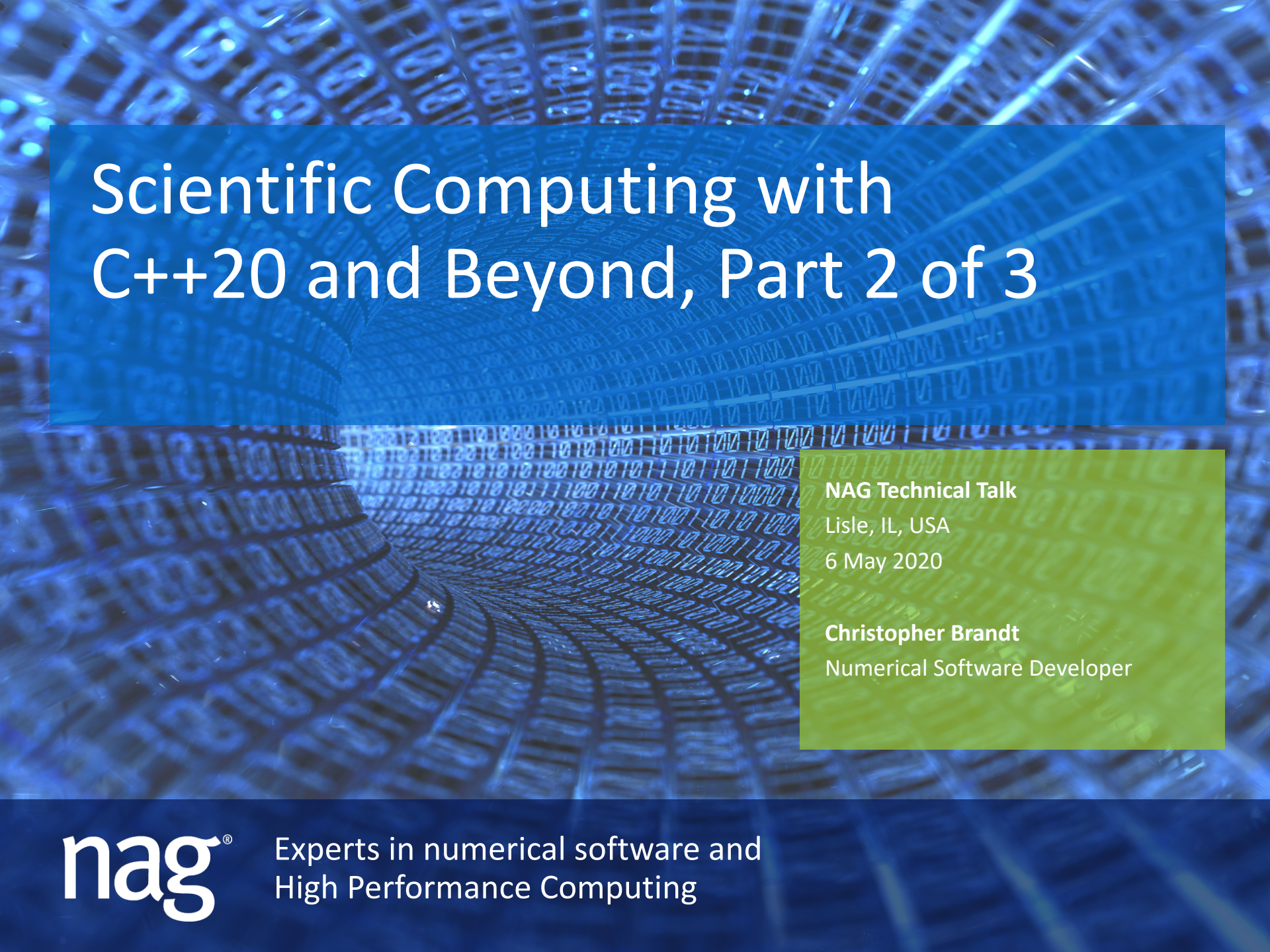


Scientific Computing with C++20 and Beyond, Part 2 of 3

NAG Technical Talk
Lisle, IL, USA
6 May 2020

Christopher Brandt
Numerical Software Developer

nag®

Experts in numerical software and
High Performance Computing

Order of Topics

- Part 1
  - Generic programming
  - concepts
  - Reference types
  - span
- Part 2
  - mdspan
  - mdarray
- Part 3
  - blas
  - linear_algebra

Co-Authors of Proposals

- National Laboratories
  - Argonne National Laboratory
  - CERN
  - Lawrence Berkeley National Laboratory
  - Los Alamos National Laboratory
  - Oak Ridge National Laboratories
  - Sandia National Laboratories
  - Swiss National Supercomputing Centre
  - U.S. Naval Research Laboratory

Co-Authors of Proposals

- Universities
  - University of British Columbia
  - University of Tennessee
- Hardware Vendors
  - AMD
  - ARM
  - Intel
  - NVIDIA

A non-owning multidimensional array reference (C++23)*

`mdspan`

mdspan – Background

- Originally introduced as array_ref (N3851) in Jan. 2014
- Formally proposed at P0009 in (Feb. 2016)
- Currently on 10th revision as of Jan 2020
- Integrated within design of other proposals
  - mdarray
  - blas
  - linear_algebra

mdspan – Motivation

- HPC-specific type
  - Used at the heart of many HPC software projects
  - Proving critical for meeting the challenges of preparing code bases for the exascale era
- Addresses concerns of performance portability
  - Data layout customization
  - Data access customization
- Enables tighter integration with:
  - Other language and Standard Library capabilities
  - Third party libraries and applications
  - User defined types

mdspan – Design

- Highly configurable
  - Ability to reduce storage to a single pointer
- Zero-overhead nature
  - Performance equivalent to using raw pointers with manual indexing
- Elements access via overloaded function call operator()
- Pass-by-value semantics
  - Cheap to construct
  - Cheap to copy
  - Trivial destructor

mdspan – Foundational Type

- Current proposals
  - mdarray
  - blas
  - linear_algebra
- Future proposals
  - Batched linear algebra
  - Machine learning
  - Audio library
  - …
- User types

mdspan – Synopsis

```
namespace std {

  // class template mdspan
  template <class ElementType,
            class Extents,
            LayoutPolicy   = layout_right,
            AccessorPolicy = accessor_basic<ElementType>>
    class basic_mdspan;

  // Type alias
  template <class T, ptrdiff_t... Extents>
    using mdspan = basic_mdspan<T, extents<Extents...>>;

}
```

mdspan – Synopsis

```
namespace std {

  // class template extents
  template <ptrdiff_t... Extents>
    class extents;

  // Layout mapping policies
  class layout_left;
  class layout_right;
  class layout_stride;

  // Accessor policies
  template <class ElementType>
    class accessor_basic;

  // class template mdspan
  // (from previous slide)

}
```

mdspan – Construction

```
// Contiguous container for constructing mdspan objects
std::vector<double> x{ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 };


// Static extents using type alias
auto m1 = std::mdspan<double, 2, 3>{ x.data() };                    // sizeof(m1) = 8

// Dynamic extents using type alias
auto m2 = std::mdspan<double,
                      std::dynamic_extent,
                      std::dynamic_extent>{ x.data(), 2, 3 }; // sizeof(m2) = 24

// Static extents using basic_mdspan
auto m3 = std::basic_mdspan<double,
                            std::extents<2, 3>,
                            std::layout_right,
                            std::accessor_basic<double>>{ x.data() };


// Element access
double v = m1(0, 1); // = 2
```

mdspan – extents<Extents...> class template

- Specifies the domain for the basic_mdspan object
- Extents may be provided statically or dynamically
- Ability to provide extents statically can help significantly with compiler optimizations

mdspan – LayoutPolicy Concept

- Maps a multi-index to a contiguous sequence of values
- mdspan includes three models of LayoutPolicy
  - layout_left (column-major ordering)
  - layout_right (row-major ordering)
  - layout_stride (non-contiguous memory)
- Customization point for users
  - Tiled layouts
  - Various forms of symmetric layouts
  - Sparse layouts
  - Compressed layouts

mdspan – AccessorPolicy Concept

- Allows you to control how memory offsets are translated into values, references, and pointers
- mdspan includes one model of AccessorPolicy
  - accessor_basic
- Customization point for users
  - Non-aliasing semantics (e.g., restrict in C)
  - Atomic access (using std::atomic_ref, C++20)
  - Access remote memory
  - Access data stored in a compressed format

mdspan – Slicing

```
// Contiguous container
std::vector<double> x( 4*4*3, 1.0 );


// mdspan object
auto m = std::basic_mdspan<double,
                           std::extents<4, 4, 3>,
                           std::layout_left>{ x.data() };


// subspan objects

auto sub1 = std::subspan(m, std::all, std::all, 1);

// std::basic_mdspan<double, std::extents<4, 4>,
//                   std::layout_left>, std::accessor_basic<double>>

auto sub2 = std::subspan(m, std::pair{0, 2}, 3, std::all);

// std::basic_mdspan<double, std::extents<-1, 4>,
//                   std::layout_stride<1, 16>, std::accessor_basic<double>>
```

mdspan – Member Functions

```
// Contiguous container
std::vector<double> v{ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 };


// mdspan object
auto ds = std::mdspan<double,
                      std::dynamic_extent,
                      std::dynamic_extent>{ y.data(), 2, 3 };


// From Extents data member
int            e1 = m.rank();           // = 2
int            e2 = m.rank_dynamic();   // = 2
std::ptrdiff_t e3 = m.static_extent(0); // = -1
std::ptrdiff_t e4 = m.static_extent(1); // = -1
std::ptrdiff_t e5 = m.extent(0);        // = 2
std::ptrdiff_t e6 = m.extent(1);        // = 3
std::ptrdiff_t e7 = m.size();           // = 6
std::ptrdiff_t e8 = m.unique_size();    // = 6


// Continued on next slide...
```

mdspan – Member Functions

```
// Continued from previous slide...


// From LayoutMapping data member
bool           m1 = m.is_unique();            // = true
bool           m2 = m.is_contiguous();        // = true
bool           m3 = m.is_strided();           // = true
bool           m4 = m.is_always_unique();     // = true
bool           m5 = m.is_always_contiguous(); // = true
bool           m6 = m.is_always_strided();    // = true
std::ptrdiff_t m7 = m.stride(0);              // = 3
std::ptrdiff_t m8 = m.stride(1);              // = 1


// Methods that return member variables
auto acc = m.accessor();
auto map = m.mapping();
auto ptr = m.data();
```

mdspan – Usage as a Function Parameter

```
// Adapted from blas proposal (P1673R2)
template <class in_vector_1_t,
          class in_vector_2_t>
auto dot(in_vector_1_t v1,
         in_vector_2_t v2)
{
  // Constraints
  assert( v1.rank()    == v2.rank()    );
  assert( v1.extent(0) == v2.extent(0) );

  // Deduce type of and initialize return value
  decltype( v1(0)*v2(0) ) result = 0;

  // Compute and return solution
  for (int i = 0; i < v1.extent(0); ++i)
    result += v1(i) * v2(i);

  return result;
}
```

mdspan – Usage as a Function Parameter

```
// Contiguous containers
std::vector<double> x{ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 };
std::vector<double> y{ 2.0, 2.0, 2.0, 2.0, 2.0, 2.0 };


// mdspan objects
auto a = std::mdspan<double, 6>{ x.data() };
auto b = std::mdspan<double, 6>{ y.data() };


// Pass mdspan objects by value to function
auto r1 = dot(a, b); // = 42
```

mdspan –Zero Overhead Demonstration

```
// From mdspan case study (SC19)
// Subspan3D Benchmark
for (std::ptrdiff_t i = 0; i < s.extent(0); ++i {
  auto sub_i = std::subspan(s, i, std::all, std::all);
  for (std::ptrdiff_t j = 0; j < s.extent(1); ++ j) {
    auto sub_i_j = std::subspan(sub_i, j, std:all);
    for (std::prtdiff_t k = 0; k < s.extent(2); ++k) {
      sum += sub_i_j(k);
    }
  }
}
```

mdspan – Implementation

- Production-Quality Reference Implementation
  - github.com/kokkos/mdspan
  - Header-only (nothing to build)
  - Compatible with
    - Clang 10.0 and GCC 9.2 (c++11, c++14, c++17, and c++2a flags)
    - GCC 10.0 (c++11, c++14, and c++17 flags)

mdspan – Reference

- C++ Standards Committee Papers
  - Multidimensional Bounds, Index and array_view (N3851)
  - mdspan: A Non-Owning Multidimensional Array Reference (P0009R10)
- Additional Reading
  - mdspan in C++: A Case Study in the Integration of Performance Portable Features Into International Language Standards (SC19)
  - Modern C++ in Computational Science (SIAM CSE 2019)

An owning multidimensional array analogue of mdspan (C++23)*

`mdarray`

mdarray – Motivation

- Cases with small, fixed-size dimensions
  - Non-owning semantics of mdspan may preclude some optimizations
  - The removal of this indirection can eliminate these inefficiencies
- Expected to serve as a foundational type within user defined types and applications

mdarray – Design

- Major goal was to parallelize to the design of mdspan as much as possible (but no more)
- Principle differences with mdspan:
  - Container semantics instead of reference semantics
    - Move constructor and move assignment operator
    - const and non-const versions of operations
  - Includes a means of interoperating with basic_mdspan
- Extents design reused
- LayoutPolicy design reused
- ContainerPolicy subsumes AccessorPolicy

mdarray – ContainerPolicy Concept

- Most difficult piece in designing mdarray
- Still under consideration
- Current design of mdarray includes two models of ContainerPolicy:
  - array_container_policy
    - Only used with all extents are static
    - Underlying data stored in std::array
  - vector_container_policy
    - Used when at least one extent is dynamic
    - Underlying data stored in std::vector

mdarray – Synopsis

```
namespace std {

  // class template mdspan
  template <class ElementType,
            class Extents,
            LayoutPolicy    = layout_right,
            ContainerPolicy = see-previous>
    class basic_mdarray;

  // Type alias
  template <class T, ptrdiff_t... Extents>
    using mdarray = basic_mdarray<T, extents<Extents...>>;

}
```

mdarray – Synopsis

```
namespace std {

  // class template extents  (reused from mdspan)

  // Layout mapping policies (reused from mdspan)

  // Container policies
  template <class ElementType, size_t N>
    class array_container_policy;
  template <class ElementType, class Allocator = std::allocator<ElementType>>
    class vector_container_policy;

  // class template mdarray
  // (from previous slide)

}
```

mdarray – Construction

```
// STL containers
std::array<double, 100> a;  // sizeof(a) = 800
std::vector<double> v(100); // sizeof(v) = 24


// mdarray objects
auto m1 = std::mdarray<double, 10, 10>{ };

auto m2 = std::mdarray<double,
                       std::dynamic_extent,
                       std::dynamic_extent>{ 10, 10 };

auto m3 = std::basic_mdarray<double,
                             std::extents<2, 3>,
                             std::layout_right,
                             std::vector_container_policy<double>>{ };


// sizeof(m1) = 800
// sizeof(m2) = 40 (std::vector + 2 dynamic extent)
// sizeof(m3) = 24 (std::vector)
```

mdarray – Integration with mdspan

```
// mdarray objects
auto m1 = std::mdarray<double, 4, 4, 3>{ };
auto m2 = std::mdarray<double, 4, 4, stdex::dynamic_extent>{ 3 };


// subspan objects
auto v1 = m1.view(); // sizeof(v1) = 8  (pointer only)

// std::basic_mdspan<double,
//                   std::extents<4, 4, 3>,
//                   std::layout_right,
//                   std::array_container_policy<double, 48>>


auto v2 = m2.view(); // sizeof(v2) = 16 (pointer + 1 dynamic extent)

// std::basic_mdspan<double,
//                   std::extents<4, 4, stdex::dynamic_extent>,
//                   std::layout_right,
//                   std::vector_container_policy<double, std::allocator<double>>>
```

mdarray – Passing a View to a Function

```
// From before
template <class in_vector_1_t,
          class in_vector_2_t>
auto dot(in_vector_1_t v1,
         in_vector_2_t v2)
{
  // Constraints
  assert( v1.rank()    == v2.rank()    );
  assert( v1.extent(0) == v2.extent(0) );

  // Deduce type of and initialize return value
  decltype( v1(0)*v2(0) ) result = 0;

  // Compute and return solution
  for (int i = 0; i < v1.extent(0); ++i)
    result += v1(i) * v2(i);

  return result;
}
```

mdarray – Passing a View to a Function

```
// mdarray objects
auto a = std::mdarray<double, 6>{ };
auto b = std::mdarray<double, 6>{ };


// Define values for each mdarray object
for (int i = 0; i < 6; ++i)
  a(i) = i + 1.0;

for (int i = 0; i < 6; ++i)
  b(i) = 2.0;


// Pass view of mdarray objects to function
auto r = dot( a.view(), b.view() ); // = 42
```

mdarray – Implementation

- Reference Implementation
  - github.com/kokkos/mdarray
  - Header only (nothing to build)
  - Compatible with
    - Clang 10.0 and GCC 9.2 (c++17 and c++2a flags)
    - GCC 10.0 (c++17 flag only)
  - Note that ContainerPolicy design still under consideration and may change

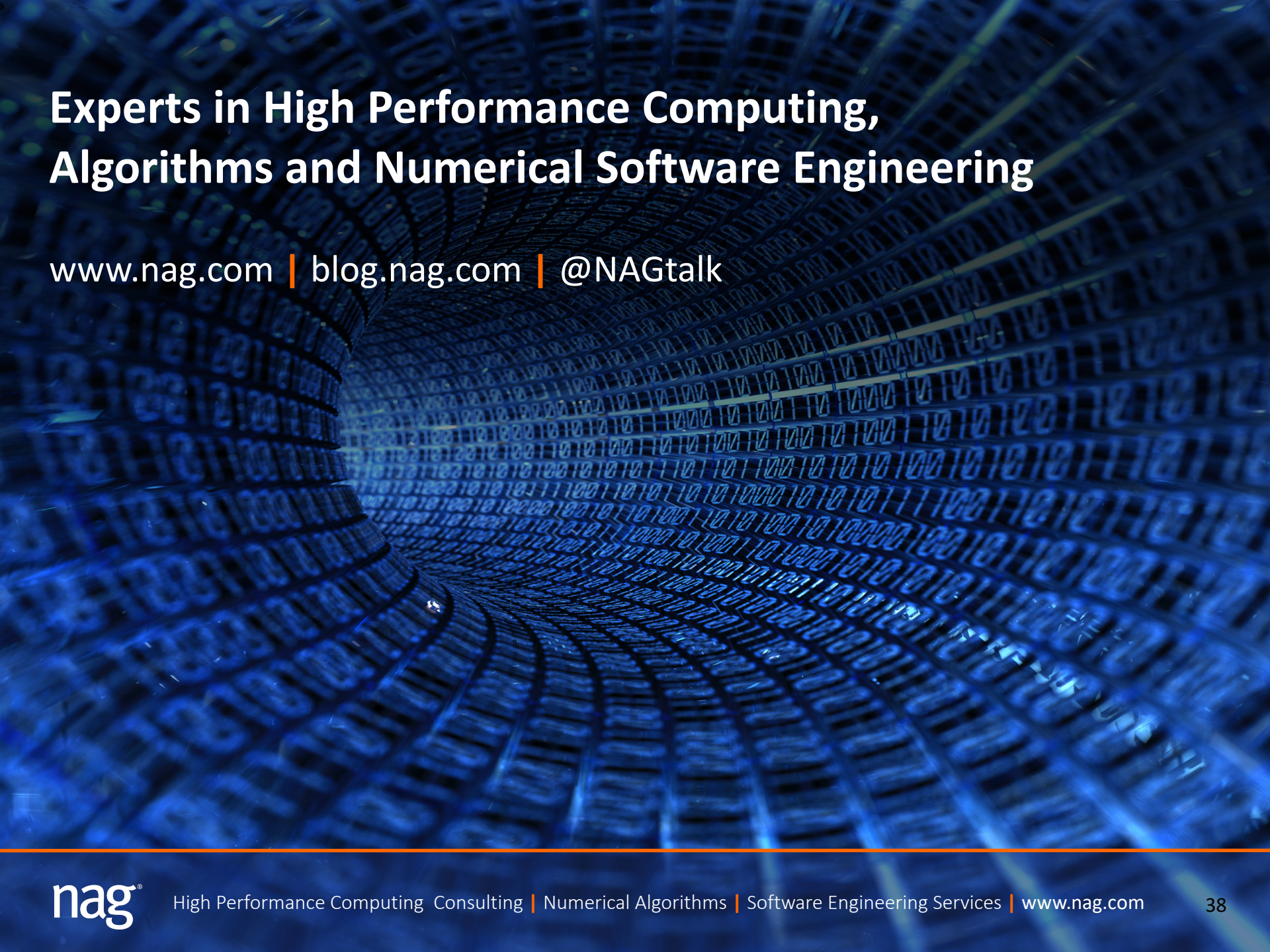
mdarray – Reference

- C++ Standards Committee Papers
  - mdarray: An Owning Multidimensional Array Analog of mdspan (P1684R0)
- Additional Reading
  - mdspan in C++: A Case Study in the Integration of Performance Portable Features Into International Language Standards (SC19)

Preview of Part 3

- blas
  - A free function linear algebra interface based on the BLAS
- linear_algebra
  - Bringing “Matlab-like” functionality to the C++ Standard Library

Questions?


Experts in High Performance Computing,
Algorithms and Numerical Software Engineering
www.nag.com | blog.nag.com | @NAGtalk