CERN School of Computing

Template Metaprogramming
for Massively Parallel Scientific Computing

*Lecture **3***

Templates for Iteration; Thread-level Parallelism

Jiří Vyskočil

Czech Technical University
AS CR – ELI Beamlines

Inverted CERN School of Computing, 29 February – 2 March 2016

CERN School of Computing

Lecture series overview

- **techniques for writing algorithms for physics computing in modern C++**
- **we will strive to produce code which is**
  - easy to read
  - efficient
  - modular
- **we will introduce some specific methods to**
  1) enable natural syntax for mathematical operations
  2) tap into the CPU's vector processing capabilities
  3) process large data sets
- **and explore their inner workings**

CERN School of Computing

Previous Lectures

Monday: Expression Templates

- by means on compile-time expansion of an expression represented by tree built using C++ template system
- generate efficient low-level code from a high level "object-oriented" syntax

Yesterday: Vectorization with Expression Templates

- SIMD instructions, data layouts (AoS, SoA, AoSoA)
- applying ET idiom to achieve vectorization while keeping the code nice and clean

CERN School of Computing

In this lecture

"process large data sets"

- the last piece we need to write efficient single-node code
- focus on large multi-dimensional arrays – these can represent scalar/vector fields, etc. in physics
- we will note that an algorithm can often be separated into "iteration" and "computation"
- we will use this separation to easily implement thread-level parallelism over our large data set
- finally we will see how we can combine elements from all the lectures to implement a working Maxwell's equations integrator with a self-explanatory code

Large multi-dimensional arrays

- **very common in physics**
- **scalar fields, vector fields, matrices (2 counts as “multi-”)**

- **no standard way to handle large N-D arrays in C++**
  - maybe post-C++17 (Ł. Mendakiewicz & H. Sutter)
- **many libraries with different approaches**
- **in this lecture, try to focus more on the concepts and reasons behind the implementation, than the presented solutions**
- ***“why” is more important than “how”***

CERN School of Computing

Separating iteration and computation

- **lots of algorithms are in the form**
  - iterate through an array
  - perform a computation on each cell, usually also using neighbouring cells, or cells from different a array

```
for (int i=1; i<N; ++i) {
  c[i] = f(a[i]) + b[i-1];
}
```

- **iteration – an operation on the “array” (or arrays)**
- **computation – an operation on the “cell” (or cells)**
- **we can treat these separately**
  - we can keep the iteration code at one place, and changes will get applied to all algorithms which use it

Functional approach

- **replace the for loop with a function which does the iteration**
- **one parameter is a reference to the "computation kernel"**
  - the function which performs the computation
- **other parameters are references to the data being manipulated**
- **the iterating function will call the computation kernel on the elements of the array (or more generally a "collection")**
- **called "map" or "apply" in functional programming**
  - we say we "map the function on the elements of the array"
  - or that we "apply the function to the elements of the array"

* the distinction between "map" and "apply" is not important for this lecture

CERN School of Computing

Functional approach

- **there are many ways to implement this**
- **C++ offers "std::transform" for collections which support STL iterators**
  - std::transform(begin, end, result, function);
  - applies **function** on elements between iterator **begin** and **end**, and writes the result to elements starting at iterator **result**
- **this methods only supports linear access pattern**
- **multi-dimensional arrays often require different treatment**
  - we want to address the dimensions separately

```
for(int i=0; i<XMAX; ++i) {
  for(int j=0; j<YMAX; ++j) {
    c[i][j] += ( a[i+1][j] + a[i][j+1] ) / (h*h);
  }
}
```

CERN School of Computing

Functional approach example

- **1st "layer of indirection" - an iterating function "apply_each"**
- **actually two overloaded "apply_each" functions**
- **the syntax of these functions will be**

apply_each(func, data)	applies function **func** taking no additional arguments to every element in **data**
apply_each(func, data, lmin, lmax)	applies **func** with no arguments to elements of **data** which lie between limits **lmin** and **lmax**

- **any additional data to be operated on will be passed by lambda capture**

Lambda expressions I

- **lambda expressions serve to create a “closure”: an unnamed function object capable of capturing variables in scope**
- **the closure can be stored in a variable, or passed as a function parameter**
- **the syntax is:** [capture-list] (params) -> ret { body }
- **params are parameters, as for a regular function**
- **ret optionally specifies the return type**
- **capture-list specifies variables to be captured form the enclosing scope**

CERN School of Computing

Lambda expressions II

- **capture-list is a comma-separated list of zero or more variables to be captured form the enclosing scope**
- **these variables will be then accessible within the lambda expression's body**
- **you can capture the variable by reference or by value**
- **you can optionally specify a "default capture" to capture all variables referenced in the lambda's body**
- **the variables are captured at the point of definition of the lambda expression**

```
int a = 5;

// default capture by value
auto f1 = [=](){ cout << a; };
// default capture by reference
auto f2 = [&](){ ++a; cout << a; };
// default by value, but a by ref.
auto f3 = [=,&a](int x)
                { a++; return x + a; }
f1();               // prints 5
f2();               // prints 6
f2();               // prints 7
f1();               // prints 5
int b = f3(5);
cout << b;          // prints 13
cout << a;          // prints 8
```

CERN School of Computing

Functional approach implementation

- **let's start with an “apply” function which will operate on 2D arrays**
- **for this presentation, we will assume we have a 2D array class which provides**
  - efficient chained operator[] to support the data[i][j] syntax
  - methods size_x() and size_y() which return the array size along respective dimensions

```
class array_2d {
  ...
};

array_2d data(x_max, y_max);

for(int i=0; i<data.size_x(); ++i)
  for(int j=0; j<data.size_y(); ++j){
    data[i][j] = i * y_max + j ;
  }
}
```

CERN School of Computing

Functional approach example I

- **our basic *apply* function takes an argument of type std::function<R> (C++11)**
  - std::function<R> is a wrapper which can store, copy, and invoke any “Callable” target
  - that includes functions, function references and pointers, lambda expression, basically anything which has an operator()
  - R is the function's signature; no need to worry about it - it will be auto-deduced
- **the *apply* function will “translate” the function call into a for loop, relaying indexes**

```
template<
  typename R, typename D
>
void apply_each(
    std::function<R> f, D&& d)
{
  for (int i=0; i<d.size.x(); ++i)
    for (int j=0; j<d.size.y(); ++j)
      f ( d, i, j );
}

template<typename T>
inline static double sqr(T& d, int i, int j) {
  d[i][j] *= d[i][j];
}

array_2d data;  // ... load initial values
apply_each(sqr, data);
```

CERN School of Computing

Functional approach example II

- **the computation kernel needs to accept the indexes as its arguments**
- **we have to pass the data to the *apply* function as we need to know their size**
- **any additional arguments can be captured by the lambda expression**

```
template<
  typename R, typename D
>
void apply_each(
    std::function<R> f, D&& d)
{
  for (int i=0; i<d.size.x(); ++i)
    for (int j=0; j<d.size.y(); ++j)
      f ( d, i, j );
}

array_2d data, a;  // ... load initial values
apply_each(
    [&a] (array_2d &d, int i, int j) {
      d[i][j] += a[i][j];
    }, data
);
```

CERN School of Computing

Functional approach example III

- **long computation kernels embedded inside the lambda's body might lead to unreadable code**
- **you can define your kernel as a function, and use the lambda to relay needed data**
- **you need to avoid the function call overhead – your kernel has to be inlined**

```
template<
  typename R, typename D
>
void apply_each(
    std::function<R> f, D&& d)
{
  for (int i=0; i<d.size.x(); ++i)
    for (int j=0; j<d.size.y(); ++j)
      f ( d, i, j );
}

template<typename T>
inline double mul(T& d, T& a, int i, int j) {
  d[i][j] *= a[i][j];
}

array_2d data;  // ... load initial values
apply_each( [&a] (array_2d d, int i, int j) {
      mul(d, a, i, j);
});
```

CERN School of Computing

Inlining I

- **inlining – replacing the function call with the function's body**
- **this eliminates the function call overhead, as the generated code doesn't actually contain any function call**
- **you can hint the compiler that you want some function to be inlined by using the *inline* keyword**
- **but – it's entirely up to the compiler to decide whether the function will be inlined or not!**

```
inline void f (
   Data& d, int i)
{
  d[i] += 2*i;
}

// the original loop
for (int i=0; i<N; ++i){
  f(d,i);
}

// actual emitted code will
// be equivalent to
for (int i=0; i<N; ++i){
  d[i] += 2*i;
}

// if the inlining did happen
```

Inlining II

- **the compiler decides whether to inline a function or not**
- **there is a trade-off between the cost of the function call overhead, and the increased executable size caused by the inlining – this can lead to misses in the instruction cache**
  - for reasonably-sized kernels, the function call overhead is always worse – think large arrays with millions of elements which would lead millions of function calls
- **what will and will not be inlined – a rule of thumb**
  - functions defined in a different compilation unit – usually a different cpp file cannot be inlined
  - member functions, and functions defined in included header files can be inlined
- **check whether the compiler did the inlining**
  - use the -Winline switch on gcc and icc for a report

CERN School of Computing

Bounded iteration I

- **with a simple modification, we can introduce a version which performs ranged iteration**
- **we include bounds as arguments to the *apply* function**
- **the kernel function stays the same**
- **now the kernel can access neighbouring cells**
  - provided we pass correct bounds
- **we will not keep this version – there is something better on the next slide**

```
template<
  typename R, typename D
>
void apply_each(
    std::function<R> f, D&& d,
    int xmin, int xmax, int ymin, int ymax)
{
  for (int i=xmin; i<xmax; ++i)
    for (int j=ymin; j<ymax; ++j)
      f ( d, i, j );
}

array_2d data;  // ... load initial values
apply_each( [](array_2d &d, int i, int j){
      d[i][j] += d[i][j-1] + d[i-1][j];
  },
  data, 5, 100, 6, 200
);
```

CERN School of Computing

Bounded iteration II

- **build a structure which holds the indexes**
- **this adds another “layer of indirection”, now on the indexing**
  - syntactically, indexing now “goes through” the “Index” structure
  - everything is resolved at compile time, so the resulting machine code is the same as the previous example
- **our kernels still stay the same**
  - but we could rewrite them to take the Index objects instead of i,j

```
struct Index {
  int x, y;
  // and a constructor taking {x, y}
};

template< typename R, typename D >
void apply_each(
    std::function<R> f, D&& d,
    Index &bmin, Index &bmax)
{
  for (int i=bmin.x; i<bmax.x; ++i)
    for (int j=bmin.y; j<bmax.y; ++j)
      f ( d, i, j );
}

Index bmin{5, 6}, bmax{100, 200};
apply_each( [](array_2d &d, int i, int j){
      d[i][j] += d[i][j-1] + d[i-1][j];
  }, data, bmin, bmax
);
```

CERN School of Computing

Bounded iteration III

- **some might not like that the lambda has to take the data as a parameter, and would prefer using only captures**
- **you can remove the “D d” parameter form the ranged version, and always pass the indexes even if you want to operate on the whole array**
- **here we implemented it together with changing the lambda syntax to only take one parameter – the Index**

```
struct Index {
  int x, y; // and a constructor taking {x, y}
};

template< typename R >
void apply_each(std::function<R> f,
    Index &bmin, Index &bmax)
{
  for (int i=bmin.x; i<bmax.x; ++i)
    for (int j=bmin.y; j<bmax.y; ++j)
      f ( {i, j} );
}

array_2d data;
Index bmin{0, 0};
Index bmax{data.size_x(), data.size_y()};
apply_each( [&data]( Index idx ){
      data[idx.x][idx.y] *= data[idx.x][idx.y];
  }, bmin, bmax
);
```

CERN School of Computing

Another indexing option

- the kernel functions in our example take the coordinates as additional parameters
- we can instead provide a proxy class “Cell” with special operator[] which would take relative coordinates
- instead of `c[i][j] = a[i-1][j] + b[i][j-1];`
- we would write `c = a[-1][0] + b[0][-1];`
  - no index means c[0][0]
- this could be achieved by using template expressions with a special operator[] definition

CERN School of Computing

Use case: switching coordinate systems

- **let's say we have some code written in cartesian coordinates**
- **we would like to implement spherical coordinates**
- **that's just a simple change in one function – the computation kernel**
- **everything else stays the same**
  - except of course the initial values, and the tools we use for data post-processing
- **no copy-pasting required except for the function definition header**

```
enum Coord {
  CART,
  SPH
};

template<Coord C> kernel(Index& idx)
{ }

template<> kernel<CART>(Index& idx)
{ /* cartesian code */ }

template<> kernel<SPH>(Index& idx)
{ /* spherical code */ }

template<Coord C>
void do_stuff() {
  Data data = { /* data intialization */ }
  apply_each ( kernel<C>, data );
}
```

CERN School of Computing

Dimension-agnostic algorithms I

- **we can provide "apply" functions for different number of dimensions**
- **add a DIM template parameter to the declaration**
- **specialize for DIM=1, 2, 3,...**
- **this code doesn't compile!**
  - you cannot partially specialize template functions

```
template< typename R, typename D, int DIM >
void apply_each<DIM>(std::function<R> f, D&& d) { }

template< typename R, typename D >
void apply_each<2>(std::function<R> f, D&& d) {
  for (int i=0; i<d.size.x(); ++i)
    for (int j=0; j<d.size.y(); ++j)
      f ( { i, j } );
}

template< typename R, typename D >
void apply_each<3>(std::function<R> f, D&& d) {
  for (int i=0; i<d.size.x(); ++i)
    for (int j=0; j<d.size.y(); ++j)
      for (int k=0; k<d.size.z(); ++k)
        f ( { i, j, k } );
}
```

CERN School of Computing

Dimension-agnostic algorithms II

- **you cannot *partially specialize* functions**
- **either we will have to go with an overload**

```
template< typename R, typename D >
void apply_each(
   std::function<R> f,
   array_2d& d)
{ .... }
```

```
template< typename R, typename D >
void apply_each(
    std::function<R> f,
    array_3d& d)
{ .... }
```

CERN School of Computing

Dimension-agnostic algorithms III

- **or we could create partially specialized classes**
  - you would then inherit form this class in classes which want to use the iteration
  - this works well if you don't mix 2D and 3D arrays in your code

```
template< int DIM > struct grid_iterator {};

template<> struct grid_iterator<2> {
  static void apply_each(std::function<R> f, D&& d)
    { ..... }
};

class computator: public grid_iterator<2> {
  array_2d a, b, c;
  void add_arrays(){
    apply_each(  [&a, &b]
                    (array_2d &d, Index idx)
      {
        d[idx.x][idx.y] = a[idx.x][idx.y] + b[idx.x][idx.y];
      },
      c
};
```

CERN School of Computing

Dimension-agnostic algorithms IV

- **there are still issues with the presented examples**
- **but let's stop here for the sake of simplicity**
- **you get the general idea on how to separate iteration from computation using functional programming**
  - iteration provided by an “apply” function
  - computation done by “computation kernels”

CERN School of Computing

Multidimensional array I

- our examples used a multidimensional array with chained operator[] subscripts
- we could implement this using template recursion, and std::vector as a storage
- instantiating the array <N> will cause “base” array <N-1> to be instantiated as well
- we provide operator[] which returns a reference to the i-th element of the N-dimensional array, which is an N-1-dimensional array

```
template<class T, int N>
struct multi_array
  : public std::vector<multi_array<T, N-1>>
{
  typedef std::vector<
      multi_array<T, N-1> > base;
  typedef typename base::value_type
      value_type;
  base v;

  auto& operator[](int i){
    return v[i];
  }
};
```

CERN School of Computing

Multidimensional array II

- **we initialize the sub-arrays using a variadic constructor**
- **we strip the actual dimension by splitting the parameters into one number head, pack the rest N-1 parameters in tail, and forward this to the lower-dimension constructor**

```
template<class T, int N>
struct multi_array
  : public std::vector<multi_array<T, N-1>>
{

  multi_array(){};

  template<typename... Args>
  multi_array(int head, Args... tail)
    : v(head)
  {
    for (int i=0; i<head; ++i){
      v[i](std::forward<Args>(tail)...);
    }
  }
};
```

CERN School of Computing

Multidimensional array III

- **we stop the recursion by specializing for N=1**
- **1-dimensional array is just the plain vector**
- **subsequent applications of operator[] will return references to vectors of N-1 dimensional containers**
- **until we reach the vector, where the operator[] returns reference to i-th value**

```
template<class T, int N>
struct multi_array {
 .....
};

template<typename T>
struct multi_array<T,1>
  : public std::vector<T>
{
  typedef std::vector<T> base;
  typedef T value_type;
  base v;

  multi_array(int n): v(n) {};
  auto& operator[](int i){
    return v[i];
  }

};
```

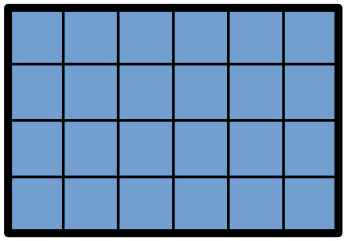
some implementation details omitted

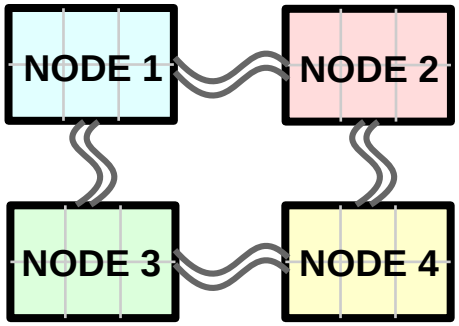
Data-parallel programming

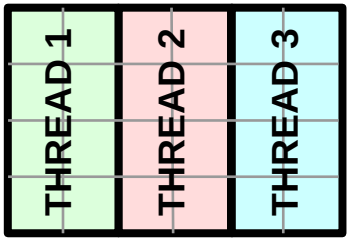
- **task parallelism**
  - different tasks performed simultaneously by independent threads on the same or different datasets
- **data parallelism**
  - the same task is performed simultaneously by many threads each operating on a subset of the dataset

Domain decomposition

- **divide the dataset into subsets which are computed independently**
  - if dependencies exist, think of the algorithm as several independent steps separated by communication steps which resolve the dependencies
- **can be performed at node-level**
  - communication by message-passing – MPI
- **or thread-level**
  - communication by reading and writing to a shared memory
  - static decomposition – each thread is pre-assigned its data subset
  - dynamic decomposition – subsets boundaries might change between iterations or even during one iteration (often difficult or practically impossible at node-level)

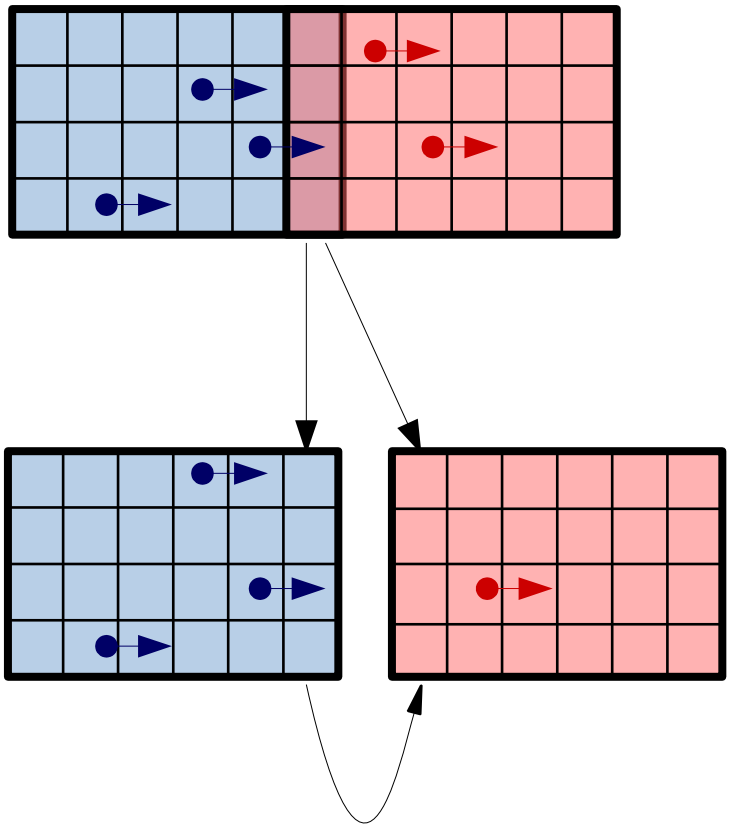






Large multi-dimensional arrays II

- **number one candidates for domain decomposition**
- **domain decomposition often maps naturally to the problem**
- **“embarrassingly parallel” algorithms**
  - those with little (or no) need for communication
  - e.g. normalizing an array of vectors – no communication needed
  - or finite-difference integration – communication on border cells only



only “halo cell” contents has to be passed to the neighbor

CERN School of Computing

Thread-level parallelism

- **use the layers-of-indirection approach**
- **do domain decomposition in the hidden implementation**
  - static
  - dynamic
- **the parallelism is implemented in one place only**
  - transparent to the interface user
  - much easier to maintain

CERN School of Computing

Write vectorization-friendly code

- **code which is well vectorized is usually easily parallelized**
- **cache optimization matters**
- **beware of aliasing (see previous lecture)**
  - when passing arrays by pointers, the target memory of the variables might overlap - “alias”
  - unlike the auto-vectorization examples where a potentially problematic loop wouldn't be vectorized, when writing thread-parallel programs you have to watch for these problems yourself

```
void f (int* offset,
        double* a,
        double* b,
        double* c)
{
  for (int i=0; i<N; ++i){
    c[i] = a[i] + b[i + (*offset)];
  }
}
```

CERN School of Computing

Loop dependencies

- **dependencies between reads and writes might lead to data races and silent failures (wrong results)**
- **more difficult than with vectorization, because we don't know which thread will execute first**

read-after-write	write-after-write	write-after-read

for(i=1; i<N; ++i) {
  a[i] = a[i-1] + b[i];
}
	for(i=0; i<N; ++i) {
	  a[i] = b[i] + c[i];
	  ...
	  a[i+2] = 2 * i;
	}
		for(i=1; i<N; ++i) {
		  a[i-1] = a[i] + c[i];
		}

writing to a variable, then reading its value	writing to the same variable in more than one iteration	possible data race – we don't know which thread will execute first
cannot be done in parallel	**proceed with extreme caution !**	**proceed with caution !**

OpenMP basics

- **OpenMP is an API for multi-threaded programming**
- **it uses a declarative approach to threading**
  - you specify regions to be parallelized by using #pragma directives
- **often used in data-parallel scenarios**
- **number of spawned threads can be controlled either at runtime, or by the environment variable OMP_NUM_THREADS**
  - it usually defaults to the number of cores in your machine, so most of the time you don't have to set anything

CERN School of Computing

OpenMP basics

```
// declaring a parallel section of code
#pragma omp parallel
{
  // this code will be executed by all threads
  #pragma omp for
  for(int i=0; i<N; ++i) {
  // work will be divided between the threads
  }
}

// a shorthand for loop parallelization
#pragma omp parallel for
for(int i=0; i<N; ++i){ ... }

// also works on random-access iterators
std::vector<float> v(N)
#pragma omp parallel for
for(auto it = begin(v); it != end(v); it++) { .... }
```

- **this is already enough to parallelize simple loops without data dependencies**
- **it will be sufficient for us today**
- **very important concepts not explained in this lecture:**
  - data scope (sharing)
  - thread scheduling

CERN School of Computing

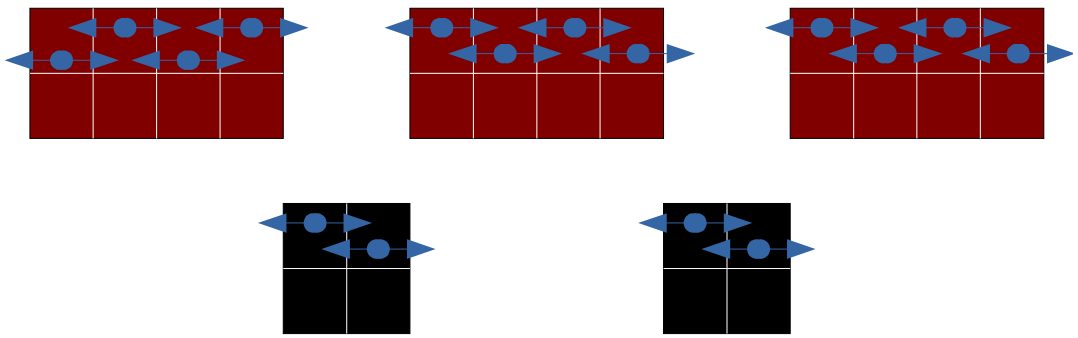
Parallelizing the “apply” function

- **we just include the OpenMP parallel for directive to the outermost loop**
- **we do this in each version of the apply_each function**
- **if we avoid calling apply_each in cases which might lead to a data race, we're fine**

```
template<
  typename Func,
  typename Data
>
void apply_each<2>(
  Func&& f, Data&& d)
{
  #pragma omp parallel for
  for (int i=0; i<d.size.x(); ++i)
    for (int j=0; j<d.size.y(); ++j)
      f ( d, Index{i, j} );
}
```

A case of concurrent writes

- **we have a function which adds to two neighbours i-1 and i+1**
- **we can schedule the threads so that they never overlap**
- **can be used to remove the WAR dependency, and some WAW dependencies**



```
void kernel(Data& c, int i, int j, Data& a,
Data& b){
  c[i-1][j]  += 2 * a[i][j];
  c[i+1][j] += 0.5 * b[i][j];
}

#if !defined(_OPENMP)
      RS = d.size.x(); BS = -1;
#endif

#pragma omp parallel for
for(int red=0; red<d.size.x()/RS; ++red)
  apply_serial(d, {red*RS, (red+1)*RS)},
                    {0, d.size.y()}, ...);

#pragma omp parallel for
for(int blk=0; blk<d.size.x()/BS; ++blk)
  apply_serial(d, {blk*BS, (blk+1)*BS)},
                    {0, d.size.y()}, ...);
```

CERN School of Computing

Putting everything together

- we'll create a solver for Maxwell's equations
- for different precisions
- with a choice of 2D or 3D
- vectorized
- OpenMP threading

Finite-Difference Time-Domain method

- **a method for integrating Maxwell's equations**
- **in each time step, we calculate for all staggered grid points:**

$$\vec{B}_{t+dt/2} = \vec{B}_{t-dt/2} + c\,dt \cdot \widetilde{rot}_{-} \vec{E}_t$$

$$\vec{E}_{t+dt} = 4\pi c\,dt \cdot \vec{J}_{t+dt/2} + c\,dt \cdot \widetilde{rot}_{+} \vec{B}_{t+dt/2}$$

- $\widetilde{rot}$ **denotes the discrete curl operator from previous slide**
- **looks familiar?**
  - it looks almost exactly like microscopic Maxwell equations “multiplied by dt” and with the curl operator replaced by the discrete curl

Templatized difference operator I

- **finite differences**
  - discrete differential in space
  - analogy to continuous derivatives
- **discrete curl**
  - assuming dx=dy=dz=1
  - shown as returning a Vector, but could be implemented as an Expression Template operation on the vector container class

```
auto inline static rot_impl (
        Vector &v, Index idx, int d){
 int i=idx.x; int j=idx.y; int k=idx.z;
 return Vector(
  v[i][j][k][2] – v[i][j+d][k][2] – v[i][j][k][1] + v[i][j][k+d][1],
  v[i][j][k][0] – v[i][j][k+d][0] – v[i][j][k][2] + v[i+d][j][k][2],
  v[i][j][k][1] – v[i+d][j][k][1] – v[i][j][k][0] + v[i][j+d][k][0]
 );
};
```

$$\nabla \times v = \left(\frac{\partial v}{\partial y} - \frac{\partial v}{\partial z}, \frac{\partial v}{\partial z} - \frac{\partial v}{\partial x}, \frac{\partial v}{\partial x} - \frac{\partial v}{\partial y} \right)$$

CERN School of Computing

Templatized difference operator II

- **finite differences**
  - discrete differential in space
  - analogy to continuous derivatives
- **think of it as replacing the partial derivative by x with subtraction of the values of two neighbouring cells in the x direction**

```
auto inline static rot_impl (
        Vector &v, Index idx, int d){
 int i=idx.x; int j=idx.y; int k=idx.z;
 return Vector(
  v[i][j][k][2] – v[i][j+d][k][2] – v[i][j][k][1] + v[i][j][k+d][1],
  v[i][j][k][0] – v[i][j][k+d][0] – v[i][j][k][2] + v[i+d][j][k][2],
  v[i][j][k][1] – v[i+d][j][k][1] – v[i][j][k][0] + v[i][j+d][k][0]
 );
};
```

$$\nabla \times v = \left(\frac{\partial v}{\partial y} - \frac{\partial v}{\partial z}, \frac{\partial v}{\partial z} - \frac{\partial v}{\partial x}, \frac{\partial v}{\partial x} - \frac{\partial v}{\partial y} \right)$$

CERN School of Computing

Templatized difference operator III

- **subtraction of the values of two neighbouring cells in the x direction**
- **to the left or to the right?**
- **both approaches might be valid**
  - details not important here
- **we call them forward and backward differences**

```
auto inline static rot_impl (
        Vector &v, Index idx, int d){
 int i=idx.x; int j=idx.y; int k=idx.z;
 return Vector(
  v[i][j][k][2] – v[i][j+d][k][2] – v[i][j][k][1] + v[i][j][k+d][1],
  v[i][j][k][0] – v[i][j][k+d][0] – v[i][j][k][2] + v[i+d][j][k][2],
  v[i][j][k][1] – v[i+d][j][k][1] – v[i][j][k][0] + v[i][j+d][k][0]
 );
};

enum CurlDirection { BCK, FWD };

template< CurlDirection dir >
inline static rot (Vector &v, Index idx) {};

auto inline static rot<BCK> (Vector &v, Index idx){
 return rot_impl(v, idx, -1); }
auto inline static rot<FWD> (Vector &v, Index idx){
 return rot_impl(v, idx, 1); }
```

Maxwell solver

- **this is the frontend of the Maxwell solver**
- **it implements the FDTD method on a staggered "Yee's grid" – the equations**

$$\vec{B}_{t+dt/2} = \vec{B}_{t-dt/2} + c\,dt \cdot \widetilde{rot}_{-} \vec{E}_t$$

$$\vec{E}_{t+dt} = 4\pi c\,dt \cdot \vec{J}_{t+dt/2} + c\,dt \cdot \widetilde{rot}_{+} \vec{B}_{t+dt/2}$$

```
using REAL = float;
const int DIM = 3;

struct data {
  container<vector<REAL, DIM>> E, B, J;
}

void integrate(){
      apply_each<DIM>(&advance_B, data);
      apply_each<DIM>(&advance_E, data);
}
void advance_B(Data &data){
  data.B[idx] += c*dt * curl<BCK>(data.E,idx);
}
void advance_E(Data &data){
  double c1 = - 4*pi*c*dt;
  data.E[idx] += c1*data.J
                 + c*dt*curl<FWD>(data.B,idx);
}
```

CERN School of Computing

Maxwell solver

- **different precisions**
  - trivially by redefining REAL
- **choice of 2D/3D**
  - redefining DIM – we already implemented dimension-independent container
- **vectorized**
  - our vector, and container were vectorized in previous lecture
- **OpenMP paralellization**
  - apply_each function was parallelized today

```
using REAL = float;
const int DIM = 3;

struct data {
  container<vector<REAL, DIM>> E, B, J;
}

void integrate(){
      apply_each<DIM>(&advance_B, data);
      apply_each<DIM>(&advance_E, data);
}
void advance_B(Data &data){
  data.B[idx] += c*dt * curl<BCK>(data.E,idx);
}
void advance_E(Data &data){
  double c1 = - 4*pi*c*dt;
  data.E[idx] += c1*data.J
                 + c*dt*curl<FWD>(data.B,idx);
}
```

CERN School of Computing

Take Away Messages

- **"iteration" and "computation" can be treated separately**
- **this separation can be expressed in functional programming paradigm as "mapping" or "applying" a computation kernel function to the data**
- **lambda expressions can be thought of as anonymous function objects which help simplify writing functional-style code – we can define them at the place where we use them**
  - they can also capture variables from the enclosing scope without the need for cumbersome specialized function declarations
- **when we separate the iteration, we can easily parallelize all loops which use these iterating function in one place using threads (OpenMP is a good choice for this use case)**
  - but we must be carefully avoid data races

CERN School of Computing

Conclusion of the Lecture Series

- **C++ templates provide a powerful system which can manipulate the syntax creating compile-time abstractions**
  - meta-program – a program that writes programs
- **combining the ET idiom with other TMP techniques can create interfaces that allow us to write natural front-end code which resolves to high-performing low-level code**
- **the abstraction can hide access to lower-level hardware features such as vectorization and threading**
- **resulting programs are portable – they use standard C++**
- **they are also easier to maintain – one group can focus solely on the backend which operates close to the hardware, the other on higher-level numerical algorithms and physics**

CERN School of Computing

Conclusion of the Lecture Series

- **the back-end code is sophisticated**

```
template <typename T, int N> class alignas(32) container {
  template<typename V, int... K>   V ret_impl(int i, std::index_sequence<K...>)  {
    return V( { std::get<K>(data)[i] ... });
  }
  Vector<T&> operator[](unsigned int i) {
    return ret_impl ( i, std::make_integer_sequence<int, N>{} );
  }
};
```

- **the front-end code is beautiful**

```
apply_each<DIM>( [&] (Index idx) {
    B[idx] += c*dt * curl<BCK>(E, idx);
});

container<vector<float, 3>>
  F = q * (E + cross(v, B));
```

CERN School of Computing

Thank you for your attention!

you can find full examples and links to additional sources at:
vysko.cz/icsc2016