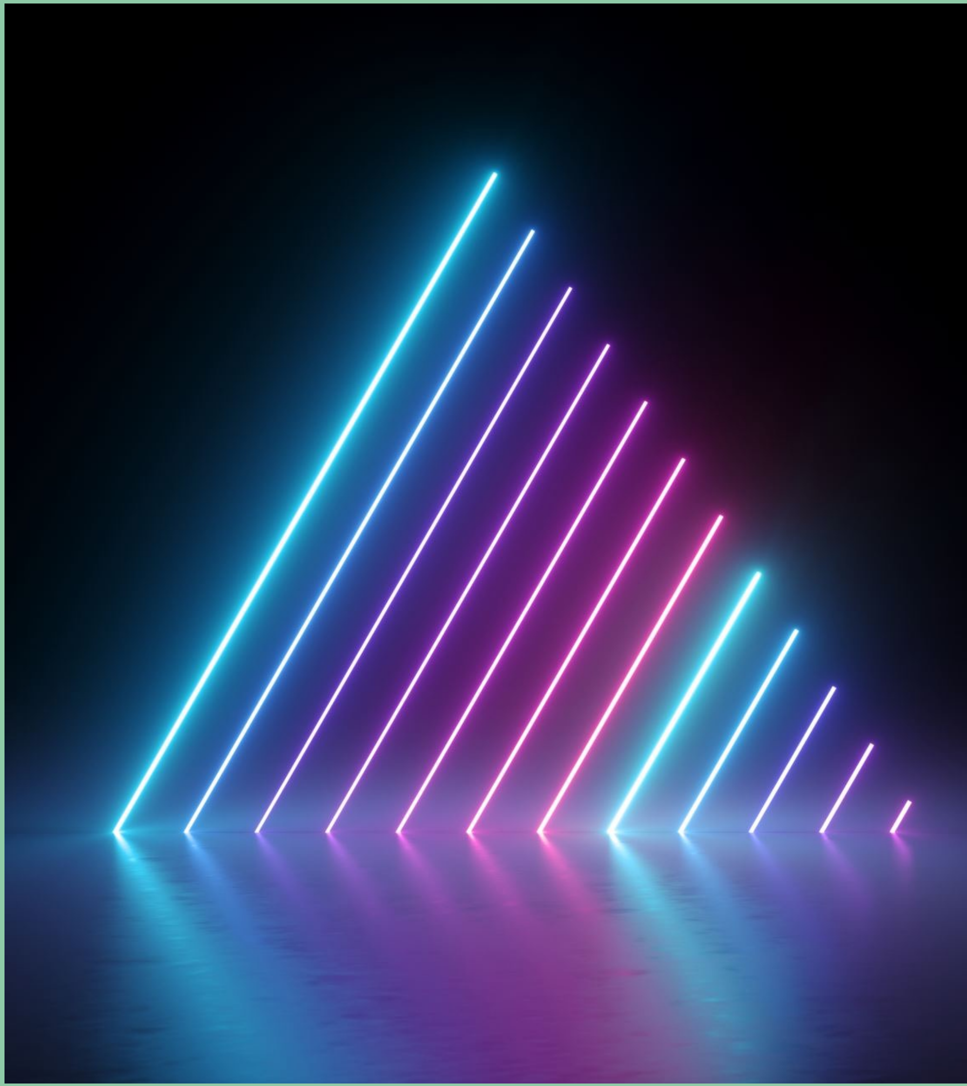
OpenMP-- just a few things about C++



CSC Summer Institute

October 9-11, 2023

- Kent Milfeld (TACC)
- Emanuele Vitali (CSC)

TACC

CSC ICT Solutions for Brilliant Minds

Using OpenMP in C++

- Using OpenMP directives and features in C++
  - Some basics
  - Range-base Iterations
  - Vector and Array Differences
  - Sophisticated Reduction Schemes

Thanks to Victor Eijkout for creating this course
and allowing me to present his work:
https://github.com/VictorEijkhout/TheArtOfHPC_vol2_parallelprogramming/tree/main/examples/omp/cxx
See also his HPC book at:
https://theartofhpc.com/pcse.html

ICT Solutions for Brilliant Minds

Output streams in parallel

- *cout* is thread safe.
- However, a line (<< x << y << z) can be broken apart at each <<.

```
void main(){
 #pragma omp parallel
 {
   int t = omp_get_thread_num();
   int s = t+1;

   cout  << s << " from " << t << endl;

 }
}
```

Output streams in parallel

- Solution, form a single string with *`stringstream`*.

```
#include <sstream>
int main(){
 #pragma omp parallel
 {
   int t = omp_get_thread_num();
   int s = t+1;
   stringstream onestring;
   onestring << s << " from " << t << endl;
   cout << onestring.str();
 }
}
```

Parallel Construct in lambda

- OpenMP parallel directives can create parallel region in lambdas.

```
int main(){
  const int s = [] () {
    int s;
    #pragma omp parallel
    #pragma omp master
    s = 2 * omp_get_num_threads();
    return s;
  } ();

  cout << s << endl;
  return 0;
}
```

TACC CSC ICT Solutions for Brilliant Minds

Class Methods: dynamic scope

- Dynamic scope holds for class methods as for any other function:

```
class C {
public:
  void class_func() {
    #pragma omp for
    for(int i=0; i<8; i++) {
      cout<< omp_get_thread_num() ;}
  }
};

int main() {
  C c;
  #pragma omp parallel num_threads(4)
  c.class_func();
}
```

OUTPUT:

23320110

(order varies)

Privatizing class members

Class members can only be privatized from (non-static) class methods.

In this example `f` can not be static:

```
// private.cxx
class foo {
private:
  int x;
public:
  void f() {
#pragma omp parallel private(x)
    somefunction(x);
  };
};
```

You can not privatize just a member:

```
// privateno.cxx
class foo { public: int x; };
int main() {
  foo thing;
#pragma omp parallel private(thing.x) // NOPE
```

Outline & questions

Parallel loops

Range based loops

Library

Questions

1. Do regular OpenMP loops look different in C++?
2. Is there a relation between OpenMP parallel loops and iterators?
3. OpenMP parallel loops vs parallel execution (algorithm) policies.

ICT Solutions for Brilliant Minds

Range syntax

Parallel loops in C++ can use range-based syntax as of OpenMP-5.0

```
// vecdata.cxx
vector<float> values(100);

#pragma omp parallel for
for ( auto& elt : values ) {
  elt = 5.f;
}

float sum{0.f};
#pragma omp parallel for reduction(+:sum)
for ( auto elt : values ) {
  sum += elt;
}
```

Performance (not shown here) is the same as C code.

C dynamic ("array") storage

- C dynamic storage use pointers:
  pointer manipulation by thread requires privatization.

```
#define Nthreads 4
int main() {
  int *array = (int*) malloc(Nthreads*sizeof(int));

  for (int i=0; i<Nthreads; i++) array[i] = 0;

  #pragma omp parallel firstprivate(array) num_threads(4)
  { int t = omp_get_thread_num();
    array += t;
    array[0] = t;
  }

  for(int i=0; i<Nthreads; i++)
      cout <<i<< " "<<array[i]<<endl;
}
```

OUTPUT:

```
0 0
1 1
2 2
3 3
```

C++ vectors

- C++ vector: copy constructor copies data
  (the same applies to an array: `int array[Nthreads];`)

```
#include <vector>
#define Nthreads 4

int main() {
  vector<int> array(Nthreads,9);

  #pragma omp parallel firstprivate(array) num_threads(4)
  { int t = omp_get_thread_num(); array[t] = t; }
  for (auto i: array) cout << i;  cout<<endl;


  #pragma omp parallel num_threads(4)
  { int t = omp_get_thread_num(); array[t] = t; }
  for (auto i: array) cout << i;  cout<<endl;

}
```

OUTPUT:

```
9999
0123
```

General concept

OpenMP can parallelize any loop over
a C++ construct that has a 'random-access' iterator.

ICT Solutions for Brilliant Minds

Ranges

```
Ranges: An abstraction of "something iterable", requiring
        begin() and end() on the range (iterable).
Views:  ranges that are defined by another range,
        that are used to transform the underlying range.
```

```
#include <ranges>
using namespace std;

int main() {
  vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8};
  auto vw = vec | views::reverse;
  cout << *vw.begin() << std::endl;
}
```

vw is a view: it does not change vec and it does not store any elements -- the creation time and size of vw is independent of the size of vec.

Ranges library | **std::ranges::zip_view**

Range access

begin	rbegin	size	empty
cbegin	crbegin	ssize	
end	rend	data	
cend	crend	cdata	

Range conversions

to (C++23)

Range primitives

range_size_t	iterator_t	range_const_reference_t (C++23)	elements_of (C++23)
range_difference_t	const_iterator_t (C++23)	range_rvalue_reference_t	
range_value_t	sentinel_t	range_common_reference_t	
range_reference_t	const_sentinel_t (C++23)		

Dangling iterator handling

dangling	borrowed_iterator_t	borrowed_subrange_t

Range concepts

range	common_range	input_range	bidirectional_range
borrowed_range	view	output_range	random_access_range
sized_range	viewable_range	forward_range	contiguous_range
constant_range (C++23)			

Views

view_interface	subrange

Range factories

empty_view	iota_view	basic_istream_view
views::empty	views::iota	views::istream
single_view	repeat_view (C++23)	
views::single	views::repeat (C++23)	

Range adaptors

views::all_t	drop_view	reverse_view	adjacent_view (C++23)
views::all	views::drop	views::reverse	views::adjacent (C++23)
ref_view	drop_while_view	as_const_view (C++23)	views::pairwise (C++23)
owning_view	views::drop_while	views::as_const (C++23)	adjacent_transform_view (C++23)
as_rvalue_view (C++23)	join_view	elements_view	views::adjacent_transform (C++23)
views::as_rvalue (C++23)	views::join	views::elements	views::pairwise_transform (C++23)
filter_view	join_with_view (C++23)	keys_view	chunk_view (C++23)
views::filter	views::join_with (C++23)	views::keys	views::chunk (C++23)
transform_view	lazy_split_view	values_view	slide_view (C++23)
views::transform	views::lazy_split	views::values	views::slide (C++23)
take_view	split_view	enumerate_view (C++23)	chunk_by_view (C++23)
views::take	views::split	views::enumerate (C++23)	views::chunk_by (C++23)
take_while_view	views::counted	**zip_view** (C++23)	stride_view (C++23)
views::take_while	common_view	**views::zip** (C++23)	views::stride (C++23)
	views::common	zip_transform_view (C++23)	cartesian_product_view (C++23)
		views::zip_transform (C++23)	views::cartesian_product (C++23)

Ranges (requires header)

The C++20 ranges library is also supported:

```
#     pragma omp parallel for reduction(+:count)
      for ( auto e : data
                | std::ranges::views::drop(1) )
        count += e;
#     pragma omp parallel for reduction(+:count)
      for ( auto e : data
                | std::ranges::views::transform
                ( []( auto e ) { return 2*e; } ) )
        count += e;
```

C++ ranges speedup

```
==== Run range on 1 threads ====
sum of vector: 50000005000000 in 6.148
sum w/ drop 1: 50000004999999 in 6.017
sum times 2   : 100000010000000 in 6.012
==== Run range on 25 threads ====
sum of vector: 50000005000000 in 0.494
sum w/ drop 1: 50000004999999 in 0.477
sum times 2   : 100000010000000 in 0.489
==== Run range on 51 threads ====
sum of vector: 50000005000000 in 0.257
sum w/ drop 1: 50000004999999 in 0.248
sum times 2   : 100000010000000 in 0.245
==== Run range on 76 threads ====
sum of vector: 50000005000000 in 0.182
sum w/ drop 1: 50000004999999 in 0.184
sum times 2   : 100000010000000 in 0.185
==== Run range on 102 threads ====
sum of vector: 50000005000000 in 0.143
sum w/ drop 1: 50000004999999 in 0.139
sum times 2   : 100000010000000 in 0.134
==== Run range on 128 threads ====
sum of vector: 50000005000000 in 0.122
sum w/ drop 1: 50000004999999 in 0.11
```

Ranges and indices

Use iota_view to obtain indices– be careful with auto:

```
                                                                 *
std::vector<int> data(N,2);

#pragma omp parallel for
for(auto  i : std::ranges::iota_view{0UZ, data.size()})
   data[i]=f(i);
std::cout<< data[0] << " " << data[N-1] << std::endl;
```

Note C++23 suffix UZ is used to type 0 as an unsigned size_t.
For older version use:

```
                                                            **
    iota_view{static_cast<size_t>(0),data.size()} )
```


* -std=c++2b
** -std=c++20

Custom iterators

Recall that

Short hand:

```
vector<float> v;
for ( auto e : v )
    ... e ...
```

for:

```
for ( vector<float>::iter
        e=v.begin();
        e!=v.end(); e++ )
    ... *e ...
```

If we want

```
for ( auto e : my_object )
    ... e ...
```

we need a sub-class for the iterator with methods such as `begin`, `end`, `*` and `++`.

Probably also `+=` and `-`

ICT Solutions for Brilliant Minds

Custom iterators

OpenMP can parallelize any range-based loop with a random-access iterator.

Class:

```
// iterator.cxx
template<typename T>
class NewVector {
protected:
  T *storage;
  int s;
public:
// iterator stuff
  class iter;
  iter begin();
  iter end();
};
```

Main:

```
NewVector<float> v(s);
#pragma omp parallel for
for ( auto e : v )
  cout << e << " ";
```

Custom iterators, exercise

Required iterator methods: **(1)**

```
NewVector<T>::iter& operator++();
T& operator*();
bool operator==( const NewVector::iter &other ) const;
bool operator!=( const NewVector::iter &other ) const;
// needed to OpenMP
int operator-
      ( const NewVector::iter& other ) const;
NewVector<T>::iter& operator+=( int add );
```

This is a little short of a full random-access iterator; the difference depends on the OpenMP implementation

Write the missing iterator methods.
Here's something to get you started. **(2)**

```
template<typename T>
class NewVector<T>::iter {
private: T *searcher;
};
template<typename T>
NewVector<T>::iter::iter( T* searcher )
  : searcher(searcher) {};
template<typename T>
NewVector<T>::iter NewVector<T>::begin() {
  return NewVector<T>::iter(storage); };
template<typename T>
NewVector<T>::iter NewVector<T>::end()   {
  return NewVector<T>::iter(storage+NewVector<T>::s); };
```

(3)

```
template<typename T>
bool  NewVector<T>::iter::operator==
      ( const NewVector<T>::iter &other ) const {
  return searcher==other.searcher; };
template<typename T>
bool  NewVector<T>::iter::operator!=
      ( const NewVector<T>::iter &other ) const {
  return searcher!=other.searcher; };
template<typename T>
NewVector<T>::iter& NewVector<T>::iter::operator++() {
  searcher++; return *this; };
template<typename T>
NewVector<T>::iter& NewVector<T>::iter::operator+=( int add )
  searcher += add; return *this; };
```

(4)

```
template<typename T>
T&  NewVector<T>::iter::operator*() {
  return *searcher; };
// needed for OpenMP
template<typename T>
int   NewVector<T>::iter::operator-
      ( const NewVector<T>::iter& other ) const {
  return searcher-other.searcher; };
```

OpenMP vs standard parallelism

Application: prime number marking (load unbalanced)

```
#pragma omp parallel for schedule(guided,8)
 for ( int i=0; i<nsize; i++) {
   results[i] = one_if_prime( number(i) );
 }
```

```
// primepolicy.cxx
transform( std::execution::par,
           numbers.begin(),numbers.end(),
           results.begin(),
           [] (int n ) -> int {
             return one_if_prime(n); }
           );
```

Standard parallelism uses Thread Building Blocks (TBB) as backend

Timings

```
==== Run primepolicy on 1 threads ====
OMP: found 0 primes; Time:       390 msec (threads= 1)
TBB: found 0 primes; Time:       392 msec
==== Run primepolicy on 25 threads ====
OMP: found 0 primes; Time:        17 msec (threads=25)
TBB: found 0 primes; Time:        19 msec
==== Run primepolicy on 51 threads ====
OMP: found 0 primes; Time:         9 msec (threads=51)
TBB: found 0 primes; Time:        13 msec
==== Run primepolicy on 76 threads ====
OMP: found 0 primes; Time:         6 msec (threads=76)
TBB: found 0 primes; Time:        15 msec
==== Run primepolicy on 102 threads ====
OMP: found 0 primes; Time:         5 msec (threads=102)
TBB: found 0 primes; Time:        71 msec
==== Run primepolicy on 128 threads ====
OMP: found 0 primes; Time:         4 msec (threads=128)
TBB: found 0 primes; Time:        55 msec
```

Reductions vs standard parallelism

Application: prime number counting (load unbalanced)

```
#pragma omp parallel for schedule(guided,8) reduction(+:prime_count)
for ( auto n : numbers ) {
  prime_count += one_if_prime( n );
}
```

```
// reducepolicy.cxx
prime_count = transform_reduce
   ( std::execution::par,
      numbers.begin(),numbers.end(),
      0,
      std::plus<>{},
      [] ( int n ) -> int {
          return one_if_prime(n); }
      );
```

ICT Solutions for Brilliant Minds

Timings

```
==== Run reducepolicy on 1 threads ====
OMP: found 9592 primes; Time:       390 msec (threads= 1)
TBB: found 9592 primes; Time:       392 msec
==== Run reducepolicy on 25 threads ====
OMP: found 9592 primes; Time:        17 msec (threads=25)
TBB: found 9592 primes; Time:        20 msec
==== Run reducepolicy on 51 threads ====
OMP: found 9592 primes; Time:         8 msec (threads=51)
TBB: found 9592 primes; Time:        13 msec
==== Run reducepolicy on 76 threads ====
OMP: found 9592 primes; Time:         6 msec (threads=76)
TBB: found 9592 primes; Time:        23 msec
==== Run reducepolicy on 102 threads ====
OMP: found 9592 primes; Time:         5 msec (threads=102)
TBB: found 9592 primes; Time:       105 msec
==== Run reducepolicy on 128 threads ====
OMP: found 9592 primes; Time:         4 msec (threads=128)
TBB: found 9592 primes; Time:        54 msec
```

Reductions

Questions

1. Are simple reductions the same as in C?
2. Can you reduce std::vector like an array?
3. Precisely what can you reduce?
4. Any interesting examples?
5. Compare reduction to native C++ mechanisms.

reductions on scalars

Same as in C,

you can now use range syntax for the loop.

```
// range.cxx
#     pragma omp parallel for reduction(+:count)
      for ( auto e : data )
        count += e;
```

ICT Solutions for Brilliant Minds

Output streams in parallel

- Arrays (a[N]) can be used in reductions: specifiedas an array section (a[:N]).
- Vectors (vector<T> v(N)) can also be used, but vectors array section (v[:N])
- are not allowed– must use a pointer to the vector

```
 int asums[N];
#pragma omp parallel for reduction(+ : asums[0:N])   ALLOWED

 vector<int> vsums(N,0);
#pragma omp parallel for reduction(+ : vsums[0:N])NOT ALLOWED
```

```
vector<int>       vsums(N,0);     //vector of sums
int *ptr_vsums = vsums.data();  //pointer to sums

#pragma omp parallel for reduction(+ : ptr_vsums[:N]) \
                          schedule (static, 1)
for (   int it=0; it<n; it++){
   for (int iv=0; iv<N; iv++)
     ptr_vsums[iv]+=iv*data[it];
}
```

ICT Solutions for Brilliant Minds

reduction on class objects

Reduction can be applied to any class for which the reduction operator is defined as `operator+` or whichever operator the case may be.

```
// reductclass.cxx
class Thing {
private:
  float x{0.f};
public:
  Thing() = default;
  Thing( float x ) : x(x) {};
  Thing operator+
      ( const Thing& other ) {
    return Thing( x + other.x );
  };
};
```

```
vector< Thing >
  things(500,Thing(1.f) );
Thing result(0.f);
#pragma omp parallel for \
    reduction( +:result )
for ( const auto& t : things )
  result = result + t;
```

A default constructor is required for the internally used init value.

Uninitialized containers

Multi-socket systems:
parallel initialization instantiates pages on sockets:
'first touch'

```
double *x = (double*)malloc( N*sizeof(double));
#pragma omp parallel for
for (int i=0; i<N; i++)
  x[i] = f(i);
```

This does not work with

```
std::vector<double> x(N);
#pragma omp parallel for
for (int i=0; i<N; i++)
  x[i] = f(i);
```

because of value initialization in the `vector` container.

Uninitialized containers (cont. 1)

Trick to create a vector of uninitialized data:

```
// heatalloc.cxx
template<typename T>
struct uninitialized {
  uninitialized() {};
  T val;
  constexpr operator T() const {return val;};
  T operator=( const T&& v ) { val = v; return val; };
};
```

so that we can create vectors that behave normally:

```
vector<uninitialized<double>> x(N),y(N);

#pragma omp parallel for
for (int i=0; i<N; i++)
  y[i] = x[i] = 0.;
x[0] = 0; x[N-1] = 1.;
```

(Question: why not use `reserve`?)

ICT Solutions for Brilliant Minds

Uninitialized containers (cont. 2)

Easy way of dealing with this:

```
template<typename T>
class ompvector : public vector<uninitialized<T>> {
public:
  ompvector( size_t s )
    : vector<uninitialized<T>>::vector<uninitialized<T>>(s) {};
};
```