Using OpenMP from C++

Victor Eijkhout

PEARC 2023

TACC

OpenMP has the opportunity to exploit features of modern C++ that are not present in C. In this course we will explore:

- range-based iteration,
- differences in treatment between vectors and arrays, and various sophisticated reduction schemes.

Basic stuff

In keeping with the desire to get rid of the C PreProcessor (CPP) in C++, a new *syntax* for OpenMP *directives* was introduced:

```
// directive.cxx
int nthreads;
[[omp::directive( parallel ) ]]
[[omp::directive( master ) ]]
nthreads = omp_get_num_threads();
```

TACC

The use of *cout* may give jumbled output:
lines can break at each <<.
Use `stringstream` to form a single stream to output.

```cpp
// hello.cxx
#pragma omp parallel
  {
    int t = omp_get_thread_num();
    stringstream proctext;
    proctext << "Hello world from " << t << '\n';
    cerr << proctext.str();
  }
```

TACC

OpenMP parallel regions can be in functions, including lambda expressions.

```
1    const int s = [] () {
2      int s;
3    #    pragma omp parallel
4    #    pragma omp master
5      s = 2 * omp_get_num_threads();
6      return s; }();
```

('Immediately Invoked Function Expression')

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

Code:

```cxx
// nested.cxx
class withnest {
public:
  void f() {
    stringstream ss;
    ss
      << omp_get_num_threads()
      << '\n';
    cout << ss str();
  };
};
int main() {
  withnest my_object;
#pragma omp parallel
  my_object.f();
```

Output:

```
executing:
    ↪OMP_MAX_ACTIVE_LEVELS=2
    ↪OMP_PROC_BIND=true
    ↪OMP_NUM_THREADS=2
    ↪./nested
2
2
```

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

In this example `f` can not be static:

```
1  // private.cxx
2  class foo {
3  private:
4    int x;
5  public:
6    void f() {
7  #pragma omp parallel private(x)
8      somefunction(x);
9    };
10 };
```

You can not privatize just a member:

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

Reference types can not be private.

Private objects are constructed/destructed by each thread.

TACC

C arrays: private pointer, but shared array:

Code:

```
1  // alloc.c
2  int *array =
3    (int*) malloc( nthreads sizeof(int));
4  for (int i=0; i nthreads; i++)
5    array[i] = 0;
6
7  #pragma omp parallel firstprivate(array)
8  {
9    int t = omp_get_thread_num();
10 // ptr arith: needs private array
11   array += t;
12   array[0] = t;
13 }
14 // ... print the array
```

Output:

```
1  Array result:
2  0:0, 1:1, 2:2, 3:3,
```

TACC

C++ vectors: copy constructor also copies data:

Code:

```
1  vector<int> array(nthreads);
2  #pragma omp parallel firstprivate(array)
3  {
4    int t = omp_get_thread_num();
5    array[t] = t+1;
6  }
7  // ... print the array
```

Output:

```
1  Array result:
2  0:0, 1:0, 2:0, 3:0,
```

Parallel loops

TACC

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 policies on algorithms.

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

```
1  // vecdata.cxx
2  vector<float> values(100);
3
4  #pragma omp parallel for
5  for ( auto& elt : values ) {
6    elt = 5.f;
7  }
8
9  float sum[0.f];
10 #pragma omp parallel for reduction(+:sum)
11 for ( auto elt : values ) {
12   sum += elt;
13 }
```

Tests show exactly the same speedup as the C code.

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

TACC

The C++20 `ranges` library is supported:

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

```
1   ==== Run range on 1 threads ====
2   sum of vector: 50000005000000 in 6.148
3   sum w/ drop 1: 50000004999999 in 6.017
4   sum times 2  : 100000010000000 in 6.012
5   ==== Run range on 25 threads ====
6   sum of vector: 50000005000000 in 0.494
7   sum w/ drop 1: 50000004999999 in 0.477
8   sum times 2  : 100000010000000 in 0.489
9   ==== Run range on 51 threads ====
10  sum of vector: 50000005000000 in 0.257
11  sum w/ drop 1: 50000004999999 in 0.248
12  sum times 2  : 100000010000000 in 0.245
13  ==== Run range on 76 threads ====
14  sum of vector: 50000005000000 in 0.182
15  sum w/ drop 1: 50000004999999 in 0.184
16  sum times 2  : 100000010000000 in 0.185
17  ==== Run range on 102 threads ====
18  sum of vector: 50000005000000 in 0.143
19  sum w/ drop 1: 50000004999999 in 0.139
20  sum times 2  : 100000010000000 in 0.134
21  ==== Run range on 128 threads ====
22  sum of vector: 50000005000000 in 0.122
23  sum w/ drop 1: 50000004999999 in 0.11
```

Use `iota_view` to obtain indices:

```
1  // iota.cxx
2  vector<long> data(N);
3  # pragma omp parallel for
4  for ( auto i : std::ranges::iota_view( 0UZ,data.size() ) )
5    data[i] = f(i);
```

Note that this uses C++23 suffix for `unsigned` `size_t`. For older versions:

```
1  iota_view( static_cast<size_t>(0),data.size() )
```

TACC

Recall that

Short hand:

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

for:

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

If we want

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

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

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

Class:

```
1  // iterator.cxx
2  template<typename T>
3  class NewVector {
4  protected:
5    T *storage;
6    int s;
7  public:
8  // iterator stuff
9    class iter;
10   iter begin();
11   iter end();
12  };
```

Main:

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

Required iterator methods:

```
1  NewVector<T>::iter& operator++();
2  T& operator*();
3  bool operator==( const NewVector::iter &other ) const;
4  bool operator!=( const NewVector::iter &other ) const;
5  // needed to OpenMP
6  int operator-
7      ( const NewVector::iter& other ) const;
8  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.

```
1   template<typename T>
2   class NewVector<T>::iter {
3   private: T *searcher;
4   };
5   template<typename T>
6   NewVector<T>::iter::iter( T* searcher )
7       : searcher(searcher) {};
8   template<typename T>
9   NewVector<T>::iter NewVector<T>::begin() {
10      return NewVector<T>::iter(storage); };
11  template<typename T>
12  NewVector<T>::iter NewVector<T>::end()   {
13      return NewVector<T>::iter(storage+NewVector<T>::s); };
```

```
1  template<typename T>
2  bool  NewVector<T>::iter::operator==
3        ( const NewVector<T>::iter &other ) const {
4    return searcher==other.searcher; };
5  template<typename T>
6  bool  NewVector<T>::iter::operator!=
7        ( const NewVector<T>::iter &other ) const {
8    return searcher!=other.searcher; };
9  template<typename T>
10 NewVector<T>::iter& NewVector<T>::iter::operator++() {
11   searcher++; return *this; };
12 template<typename T>
13 NewVector<T>::iter& NewVector<T>::iter::operator+=( int add ) {
14   searcher += add; return *this; };
```

```
1  template<typename T>
2  T&  NewVector<T>::iter::operator*() {
3    return *searcher; };
4  // needed for OpenMP
5  template<typename T>
6  int   NewVector<T>::iter::operator-
7      ( const NewVector<T>::iter& other ) const {
8    return searcher-other.searcher; };
```

TACC

Application: prime number marking (load unbalanced)
missing snippet markprimeomp
missing snippet markprimecpp

Standard parallelism uses Thread Building Blocks (TBB) as backend

Timing

TACC

```
 1   Threads:   1
 2   TBB:  Time:      392 msec
 3   Stat: Time:      389 msec
 4   Dyn:  Time:      390 msec
 5
 6   Threads: 25
 7   TBB:  Time:       20 msec
 8   Stat: Time:       32 msec
 9   Dyn:  Time:       17 msec
10
11   Threads: 51
12   TBB:  Time:       13 msec
13   Stat: Time:       15 msec
14   Dyn:  Time:        9 msec
15
16   Threads: 76
17   TBB:  Time:       29 msec
18   Stat: Time:       11 msec
19   Dyn:  Time:        6 msec
20
21   Threads: 102
22   TBB:  Time:       61 msec
23   Stat: Time:        8 msec
```

TACC

Application: prime number counting (load unbalanced)
missing snippet reduceprimeomp
missing snippet reduceprimecpp

```
 1  Threads:    1
 2  TBB:   Time:        391 msec
 3  Stat:  Time:        390 msec
 4  Dyn:   Time:        389 msec
 5
 6  Threads:   25
 7  TBB:   Time:         20 msec
 8  Stat:  Time:         17 msec
 9  Dyn:   Time:         17 msec
10
11  Threads:   51
12  TBB:   Time:         13 msec
13  Stat:  Time:          9 msec
14  Dyn:   Time:          8 msec
15
16  Threads:   76
17  TBB:   Time:         14 msec
18  Stat:  Time:          8 msec
19  Dyn:   Time:          5 msec
20
21  Threads:  102
22  TBB:   Time:         76 msec
23  Stat:  Time:          5 msec
```

Reductions

TACC

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 reductions to native C++ mechanisms.

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;
```

Use the `data` method to extract the array on which to reduce. However, this does not work:

```
1   vector<float> x;
2   #pragma omp parallel reduction(+:x.data())
```

because the reduction clause wants a variable, not an expression, for the array, so you need an extra bare pointer:

```
1   // reductarray.cxx
2   vector<int> data(nthreads,0);
3   int *datadata = data.data();
4   #pragma omp parallel for schedule(static,1) \
5     reduction(+:datadata[:nthreads])
```

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

```
1  // reductclass.cxx
2  class Thing {
3  private:
4    float x{0.f};
5  public:
6    Thing() = default;
7    Thing( float x ) : x(x) {};
8    Thing operator+
9      ( const Thing& other ) {
10     return Thing( x + other.x );
11   };
12 };
```

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

A default constructor is required for the internally used init value; see figure 34.

Reduction of four items on two threads. `i` is the OpenMP initialization, and `u` is the user initialization; each `p` stands for a partial reduction value.
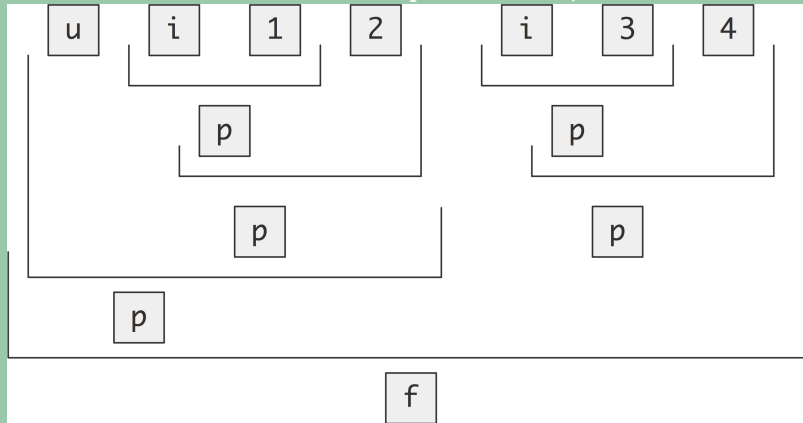
```
1    #pragma omp declare reduction
2    ( identifier : typelist : combiner )
3    [initializer(initializer expression)]
4
```

TACC

Support for *C++ iterators*

```
1  #pragma omp declare reduction \
2      (merge              // identifier
3      : std::vector<int>  // typelist
4      : omp_out.insert(omp_out.end(), omp_in.begin(),
5                        omp_in.end())  // combiner
6      )
```

You can use lambda expressions in the explicit expression for a declared reduction:

```cxx
// reductexpr.cxx
#pragma omp declare reduction\
  (minabs : int : \
  omp_out = \
      [] (int x,int y) -> int { \
        return abs(x) > abs(y) ? abs(y) : abs(x); } \
    (omp_in,omp_out) ) \
  initializer (omp_priv=limit::max())
```

You can not assign the lambda expression to a variable and use that, because `omp_in`/`out` are the only variables allowed in the explicit expression.

TACC

Count which elements fall into what bin:

```
1    for ( auto e : some_range )
2      histogram[ value(e)]++;
3
```

Collisions are possible, but unlikely, so critical section is very inefficient

TACC

Declare a reduction on a histogram object;
each thread gets a local map:

```
1  /*
2   * Reduction loop in main program
3   */
4  bincounter<char> charcount;
5  #pragma omp parallel for reduction(+ : charcount)
6  for ( int i=0; i<text.size(); i++ )
7    charcount.inc( text[i] );
```

Q: why does the inc not have to be atomic?

Give the class a `+=` operator to do the combining:

```cxx
// charcount.cxx
template<typename key>
class bincounter : public map<key,int> {
public:
// merge this with other map
  void operator+=
      ( const bincounter<key>& other ) {
    for ( auto [k,v] : other )
      if ( map<key,int>::contains(k) )
        this->at(k) += v;
      else
        this->insert( {k,v} );
  };
// insert one char in this map
  void inc(char k) {
    if ( map<key,int>::contains(k) )
      this->at(k) += 1;
    else
      this->insert( {k,1} );
  };
};
```

Use atomics because there is no reduction mechanism:

```
// mapreduceatomic.cxx
class CharCounter : public array<atomic<int>,26> {
public:
  CharCounter() {
    for ( int ic=0; ic<26; ic++ )
      (*this)[ic] = 0;
  };
// insert one char in this map
  void inc(char k) {
    if (k==' ') return;
    int ik = k-'a';
    (*this)[ik]++;
  };
};
```

OpenMP reduction on `array<int,26>`:

```
1  Using atomics    on 1 threads: time= 20.19 msec
2  OpenMP reduction on 1 threads: time= 1.966 msec
3  Using atomics    on 5 threads: time= 315.855 msec
4  OpenMP reduction on 5 threads: time= 0.52 msec
5  Using atomics    on 10 threads: time= 91.968 msec
6  OpenMP reduction on 10 threads: time= 0.364 msec
7  Using atomics    on 30 threads: time= 249.171 msec
8  OpenMP reduction on 30 threads: time= 0.556 msec
9  Using atomics    on 50 threads: time= 164.177 msec
10 OpenMP reduction on 50 threads: time= 0.904 msec
```

Make an OpenMP parallel version of:

```
1  intcounter primecounter;
2  for ( auto n : numbers )
3    if ( isprime(n) )
4      primecounter.add(n);
```

where `primecounter` contains a `map<int,int>`.

Use skeleton: mapreduce.cxx

TACC

The sequential code is as follows:

```
1  vector<int> data(100);
2  // fil the data
3  vector<int> filtered;
4  for ( auto e : data ) {
5    if ( f(e) )
6      filtered.push_back(e);
7  }
```

Let each thread have a local array, and then to concatenate these:

```
1  #pragma omp parallel
2  {
3    vector<int> local;
4  # pragma omp for
5    for ( auto e : data )
6      if ( f(e) ) local.push_back(e);
7    filtered += local;
8  }
```

where we have used an append operation on vectors:

```
1  // filterreduct.cxx
2  template<typename T>
3  vector<T>& operator+=( vector<T>& me, const vector<T>& other ) {
4    me.insert( me.end(),other.begin(),other.end() );
5    return me;
6  };
```

TACC

We could use the plus-is operation to declare a reduction:

```
1  #pragma omp declare reduction\
2    (                      \
3            +:vector<int>:omp_out += omp_in \
4            ) \
5    initializer( omp_priv = vector<int>{} )
```

Problem: OpenMP reductions can not be declared non-commutative, so the contributions from the threads may not appear in order.

Code:

```
1  #pragma omp parallel \
2    reduction(+ : filtered)
3    {
4      vector<int> local;
5  #    pragma omp for
6      for ( auto e : data )
7        if ( f(e) )
8          local push_back(e);
9      filtered += local;
10   }
```

Output:

```
1  Mod 5: 80 85 90 95
       ↪100 5 10 15 20
       ↪25 30 35 40 45
       ↪50 55 60 65 70
       ↪75
```

TACC

Parallel region, without for:

Code:

```
1  // filtertask.cxx
2  vector<int> filtered;
3  int ithread=0;
4  #pragma omp parallel
5  {
6    vector<int> local;
7    int threadnum = omp_get_thread_num();
8  #    pragma omp for
9    for ( auto e : data )
10     if ( e%5==0 )
11         local.push_back(e);
12 // create task to add local to filtered
```

Output:

```
1  Mod 5: 5 10 15 20 25
        ↪30 35 40 45 50
        ↪55 60 65 70 75
        ↪80 85 90 95 100
```

TACC

The task spins until it's its turn:

Code:

```
#    pragma omp task \
        shared(filtered,ithread)
        {
// wait your turn
        while (threadnum>ithread) {
#        pragma omp taskyield
        }
// merge
        filtered += local;
        ithread++;
        }
```

Output:

```
Mod 5: 5 10 15 20 25
       ↪30 35 40 45 50
       ↪55 60 65 70 75
       ↪80 85 90 95 100
```

TACC

You can reduce with a templated function if you put both the declaration and the reduction in the same templated function:

```
1   template<typename T>
2   T generic_reduction( vector<T> tdata ) {
3   #pragma omp declare reduction                                    \
4     (rwzt:T:omp_out=reduce_without_zero<T>(omp_out,omp_in))         \
5     initializer(omp_priv=-1.f)
6
7     T tmin = -1;
8   #pragma omp parallel for reduction(rwzt:tmin)
9     for (int id=0; id<tdata.size(); id++)
10      tmin = reduce_without_zero<T>(tmin,tdata[id]);
11    return tmin;
12  };
```

which is then called with specific data:

```
1   auto tmin = generic_reduction<float>(fdata);
```

More topics

TACC

The new C++ `random` header has a threadsafe generator, by virtue of the statement in the standard that no STL object can rely on global state. The usual idiom can not be made threadsafe because of the initialization:

```
static random_device rd;
static mt19937 rng(rd);
```

However, the following works:

```
// privaterandom.cxx
static random_device rd;
static mt19937 rng;
#pragma omp threadprivate(rd)
#pragma omp threadprivate(rng)

int main() {

#pragma omp parallel
  rng = mt19937(rd());
```

Based on the previous note, you can use the generator safely and
independently:

```cpp
1    #pragma omp parallel
2      {
3        stringstream res;
4        uniform_int_distribution<int> percent(1, 100);
5        res << "Thread " << omp_get_thread_num() << ": " << percent(rng)
           ↪<< "\n";
6        cout << res.str();
7      }
```

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

```
1    double *x = (double*)malloc( N*sizeof(double));
2    #pragma omp parallel for
3    for (int i=0; i<N; i++)
4      x[i] = f(i);
5
```

This does not work with

```
1    std::vector<double> x(N);
2    #pragma omp parallel for
3    for (int i=0; i<N; i++)
4      x[i] = f(i);
5
```

because of value initialization in the `vector` container.

TACC

Trick to create a vector of uninitialized data:

```cpp
// 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:

```cpp
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*?)

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) {};
};
```

Atomic scalar updates

Can you atomically update scalars?

- Make an object that has data plus a lock;
- Disable copy and copy-assignment operators;
- Destructor does `omp_destroy_lock`;
- Overload arithmetic operator.

(Quick self-test: why lock, not critical?)

TACC

```cpp
// lockobject.cxx
class atomic_int {
private:
  omp_lock_t the_lock;
  int _value{0};
public:
  atomic_int() {
    omp_init_lock(&the_lock);
  };
  atomic_int( const atomic_int& )
      = delete;
  atomic_int& operator=( const atomic_int& )
      = delete;
  ~atomic_int() {
    omp_destroy_lock(&the_lock);
  };
```

TACC

```
int operator +=( int i ) {
// atomic increment
  omp_set_lock(&the_lock);
  _value += i; int rv = _value;
  omp_unset_lock(&the_lock);
  return rv;
};
```

TACC

```cpp
atomic_int my_object;
vector<std::thread> threads;
for (int ithread=0;
      ithread<NTHREADS;
      ithread++) {
  threads.push_back
    ( std::thread(
      [=,&my_object] () {
        for (int iop=0; iop<nops; iop++)
          my_object += 1; } ) );
}
for ( auto &t : threads )
  t.join();
```

Timing comparison on simplest case:

Object with built-in lock:

```
1   atomic_int my_object;
2   vector<std::thread> threads;
3   for (int ithread=0;
4       ithread<NTHREADS;
5       ithread++) {
6     threads.push_back
7       ( std::thread(
8         [=,&my_object] () {
9           for (int iop=0;
        iop<nops; iop++)
10            my_object += 1; } ) );
11  }
12  for ( auto &t : threads )
13    t.join();
```

Native C++ atomics:

```
1   std::atomic<int> my_object{0};
2   #pragma omp parallel for
3   for ( size_t update=0;
4       update<NTHREADS*nops;
5       update++) {
6     my_object += 1;
7   }
8   result = my_object;
```

Native solution is 10x faster.

TACC

```
1    #include <new>
2
3    #ifdef __cpp_lib_hardware_interference_size
4    const int spread = std::hardware_destructive_interference_size
5              / sizeof(datatype);
6    #else
7    const int spread = 8;
8    #endif
9
10   vector<datatype> k(nthreads*spread);
11   #pragma omp parallel for schedule( static, 1 )
12   for ( datatype i = 0; i < N; i++ ) {
13     k[ (i%nthreads) * spread ] += 2;
14
```

Since C++17

Beware vector-of-bool!

Does not compile:

```
// boolrange.cxx
vector<bool> bits(1000000);
for ( auto& b : bits )
  b = true;
```

More subtle:

Code:

```
// booliter.cxx
vector<bool> bits(3000000);
#pragma omp parallel for schedule(static,4)
for ( int i=0; i<bits.size(); i++ )
  bits[i] = ( i%3==0 );
// and then count the million 1s
```

Output:

```
#threads=1; should be
    ↪million: 1000000
#threads=2; should be
    ↪million: 1000000
#threads=3; should be
    ↪million: 999964
#threads=4; should be
    ↪million: 999659
```

Different `bits[i]` are falsely shared.

```
1    cmake_minimum_required( VERSION 3.20 )
2    project( ${PROJECT_NAME} VERSION 1.0 )
3
4    find_package( OpenMP )
5    if( OpenMP_CXX_FOUND )
6    else()
7            message( FATAL_ERROR "Could not find OpenMP" )
8    endif()
9
10   add_executable( ${PROJECT_NAME} ${PROJECT_NAME}.cxx )
11   target_compile_features( ${PROJECT_NAME} PRIVATE cxx_std_20 )
12   target_link_libraries( ${PROJECT_NAME} PUBLIC OpenMP::OpenMP_CXX )
13
14   install( TARGETS ${PROJECT_NAME} DESTINATION . )
```