Using OpenMP from C++

Victor Eijkhout

PEARC 2023

Justification

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

Output streams in parallel

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

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

Parallel regions in lambdas

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

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

('Immediately Invoked Function Expression')

Dynamic scope for class methods

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

Code:

```
// 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
```

Privatizing class members

TACC

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
```

Vectors are copied, unlike arrays, 1

C arrays: private pointer, but shared array:

Code:

```
// alloc.c
int *array =
  (int*) malloc(nthreads*sizeof(int));
for (int i=0; i<nthreads; i++)
  array[i] = 0;

#pragma omp parallel firstprivate(array)
{
  int t = omp_get_thread_num();
// ptr arith: needs private array
  array += t;
  array[0] = t;
}
// ... print the array
```

Output:

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

Vectors are copied, unlike arrays, 2

C++ vectors: copy constructor also copies data:

Code:

```
// alloc.cxx
vector<int> array(nthreads);

#pragma omp parallel firstprivate(array)
{
  int t = omp_get_thread_num();
  array[t] = t+1;
}
// ... print the array
```

Output:

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

Parallel loops

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.

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;
}
```

Tests not reported here show exactly the same speedup as the C code.

General idea

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

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:

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

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

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

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 -

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 << " ";
```

Required iterator methods:

```
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.

Custom iterators, exercise

Write the missing iterator methods.
Here's something to get you started.

```
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); };
```

Custom iterators, solution

```
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; };
```

Custom iterators, solution

```
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

TACC

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

Timing

```
==== 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
```

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); }
    );
```

Timing

```
==== 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 reductions to native C++ mechanisms.

Scalar reductions

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;
```

Reductions on vectors

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

```
vector<float> x;
#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:

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

Reduction on class objects

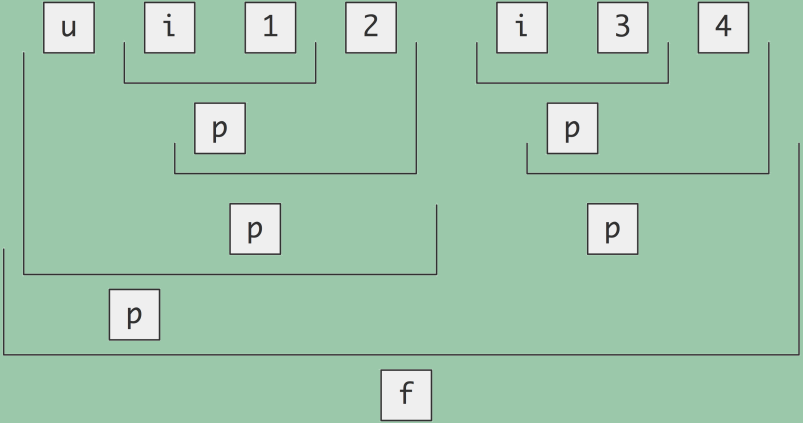
Reduction can be applied to any class for which the reduction operator is defined as `operator+` or whichever operator the case may be.

```
// 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; see figure 32.

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.



User-defined reductions, syntax

```
#pragma omp declare reduction
( identifier : typelist : combiner )
[initializer(initializer-expression)]
```

Reduction over iterators

Support for *C++ iterators*

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

You can use lambda expressions in the explicit expression:

```
// 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.

Example category: histograms

Count which elements fall into what bin:

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

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

Histogram: intended main program

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

```
bincounter<char> charcount;
#pragma omp parallel for reduction(+ : charcount)
for ( int i=0; i<text.size(); i++ )
  charcount.inc( text[i] );
```

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

Histogram solution: reduction operator

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

```
// 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} );
  };
};
```

Histogram in native C++

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]++;
  };
};
```

Histogram in native C++, comparison

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

```
Using atomics     on 1 threads: time= 20.19 msec
OpenMP reduction on 1 threads: time= 1.966 msec
Using atomics     on 5 threads: time= 315.855 msec
OpenMP reduction on 5 threads: time= 0.52 msec
Using atomics     on 10 threads: time= 91.968 msec
OpenMP reduction on 10 threads: time= 0.364 msec
Using atomics     on 30 threads: time= 249.171 msec
OpenMP reduction on 30 threads: time= 0.556 msec
Using atomics     on 50 threads: time= 164.177 msec
OpenMP reduction on 50 threads: time= 0.904 msec
```

Exercise: mapreduce

Make an OpenMP parallel version of:

```
intcounter primecounter;
for ( auto n : numbers )
  if ( isprime(n) )
    primecounter.add(n);
```

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

Use skeleton: `mapreduce.cxx`

Example category: list filtering

The sequential code is as follows:

```
vector<int> data(100);
// fil the data
vector<int> filtered;
for ( auto e : data ) {
  if ( f(e) )
    filtered.push_back(e);
}
```

List filtering, solution 1

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

```
#pragma omp parallel
{
  vector<int> local;
# pragma omp for
  for ( auto e : data )
    if ( f(e) ) local.push_back(e);
  filtered += local;
}
```

where we have used an append operation on vectors:

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

List filtering, not quite solution 2

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

```
#pragma omp declare reduction\
  (      \
        +:vector<int>:omp_out += omp_in \
        ) \
  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:

```
#pragma omp parallel \
  reduction(+ : filtered)
  {
    vector<int> local;
#   pragma omp for
    for ( auto e : data )
      if ( f(e) )
        local.push_back(e);
    filtered += local;
  }
```

Output:

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

List filtering, task-based solution

Parallel region, without for:

Code:

```
// filtertask.cxx
vector<int> filtered;
int ithread=0;
#pragma omp parallel
{
  vector<int> local;
  int threadnum = omp_get_thread_num();
#   pragma omp for
  for ( auto e : data )
    if ( e%5==0 )
        local.push_back(e);
// create task to add local to filtered
```

Output:

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

List filtering, task-based solution'

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
```

Templated reductions

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

```
template<typename T>
T generic_reduction( vector<T> tdata ) {
#pragma omp declare reduction                              \
  (rwzt:T:omp_out=reduce_without_zero<T>(omp_out,omp_in))  \
  initializer(omp_priv=-1.f)

  T tmin = -1;
#pragma omp parallel for reduction(rwzt:tmin)
  for (int id=0; id<tdata.size(); id++)
    tmin = reduce_without_zero<T>(tmin,tdata[id]);
  return tmin;
};
```

which is then called with specific data:

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

More topics

Threadprivate random number generators

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());
```

Threadprivate random use

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

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

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, 2

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`?)

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?)

Atomic updates: class with OMP lock

```
// 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);
  };
```

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

Atomic updates: usage

```
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();
```

Atomic updates, comparison to native

Timing comparison on simplest case:

Object with built-in lock:

```
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();
```

Native C++ atomics:

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

Native solution is 10x faster.

False sharing prevention

TACC

```
#include <new>

#ifdef __cpp_lib_hardware_interference_size
const int spread = std::hardware_destructive_interference_size
         / sizeof(datatype);
#else
const int spread = 8;
#endif

vector<datatype> k(nthreads*spread);
#pragma omp parallel for schedule( static, 1 )
for ( datatype i = 0; i < N; i++ ) {
  k[ (i%nthreads) * spread ] += 2;
```

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.

CMake

```
cmake_minimum_required( VERSION 3.20 )
project( ${PROJECT_NAME} VERSION 1.0 )

find_package(OpenMP)
if(OpenMP_CXX_FOUND)
else()
        message( FATAL_ERROR "Could not find OpenMP" )
endif()

add_executable( ${PROJECT_NAME} ${PROJECT_NAME}.cxx )
target_compile_features( ${PROJECT_NAME} PRIVATE cxx_std_20 )
target_link_libraries( ${PROJECT_NAME} PUBLIC OpenMP::OpenMP_CXX )

install( TARGETS ${PROJECT_NAME} DESTINATION . )
```