

# Using OpenMP from C++

Victor Eijkhout

PEARC 2023



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



The use of `cout` may give jumbled output:  
lines can break at each `<<`.

Use `stringstream` to form a single stream to output.

```
1 // hello.cxx
2 #pragma omp parallel
3 {
4     int t = omp_get_thread_num();
5     stringstream proctext;
6     proctext << "Hello world from " << t;
7     cerr << proctext.str() << '\n';
8 }
```



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:

```
1 // nested.cxx
2 class withnest {
3 public:
4     void f() {
5         stringstream ss;
6         ss
7             << omp_get_num_threads()
8             << '\n';
9         cout << ss.str();
10    };
11 };
12 int main() {
13     withnest my_object;
14     #pragma omp parallel
15     my_object.f();
```

Output:

```
1  executing:
    ↪OMP_MAX_ACTIVE_LEVELS=2
    ↪OMP_PROC_BIND=true
    ↪OMP_NUM_THREADS=2
    ↪./nested
2  2
3  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
```



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,
```





C++ vectors: copy constructor also copies data:

Code:

```
1 // alloc.cxx
2 vector<int> array(nthreads);
3
4 #pragma omp parallel firstprivate(array)
5 {
6     int t = omp_get_thread_num();
7     array[t] = t+1;
8 }
9 // ... print the array
```

Output:

```
1 Array result:
2 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.



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



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



The C++20 ranges library is also 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(
    ↪static_cast<size_t>(0),data.size() ) )
5     data[i] = f(i);
```





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; };
```



Application: prime number marking (load unbalanced)

```
1  #pragma omp parallel for schedule(guided,8)
2  for ( int i=0; i<nsize; i++) {
3      results[i] = one_if_prime( number(i) );
4  }
```

```
1  // primepolicy.cxx
2  transform( std::execution::par,
3             numbers.begin(),numbers.end(),
4             results.begin(),
5             [] (int n ) -> int {
6                 return one_if_prime(n); }
7             );
```

Standard parallelism uses Thread Building Blocks (TBB) as backend



```
1  ==== Run primepolicy on 1 threads ====
2  OMP: found 0 primes; Time:          390 msec (threads= 1)
3  TBB: found 0 primes; Time:          392 msec
4  ==== Run primepolicy on 25 threads ====
5  OMP: found 0 primes; Time:           17 msec (threads=25)
6  TBB: found 0 primes; Time:           19 msec
7  ==== Run primepolicy on 51 threads ====
8  OMP: found 0 primes; Time:           9 msec (threads=51)
9  TBB: found 0 primes; Time:          13 msec
10 ==== Run primepolicy on 76 threads ====
11 OMP: found 0 primes; Time:           6 msec (threads=76)
12 TBB: found 0 primes; Time:          15 msec
13 ==== Run primepolicy on 102 threads ====
14 OMP: found 0 primes; Time:           5 msec (threads=102)
15 TBB: found 0 primes; Time:          71 msec
16 ==== Run primepolicy on 128 threads ====
17 OMP: found 0 primes; Time:           4 msec (threads=128)
18 TBB: found 0 primes; Time:          55 msec
```





Application: prime number counting (load unbalanced)

```
1  #pragma omp parallel for schedule(guided,8) reduction(+:prime_count)
2  for ( auto n : numbers ) {
3      prime_count += one_if_prime( n );
4  }
```

```
1  // reducepolicy.cxx
2  prime_count = transform_reduce
3      ( std::execution::par,
4        numbers.begin(), numbers.end(),
5        0,
6        std::plus<>{},
7        [] ( int n ) -> int {
8            return one_if_prime(n); }
9        );
```



```
1  ==== Run reducepolicy on 1 threads ====
2  OMP: found 9592 primes; Time:      390 msec (threads= 1)
3  TBB: found 9592 primes; Time:      392 msec
4  ==== Run reducepolicy on 25 threads ====
5  OMP: found 9592 primes; Time:       17 msec (threads=25)
6  TBB: found 9592 primes; Time:       20 msec
7  ==== Run reducepolicy on 51 threads ====
8  OMP: found 9592 primes; Time:        8 msec (threads=51)
9  TBB: found 9592 primes; Time:       13 msec
10 ==== Run reducepolicy on 76 threads ====
11 OMP: found 9592 primes; Time:         6 msec (threads=76)
12 TBB: found 9592 primes; Time:       23 msec
13 ==== Run reducepolicy on 102 threads ====
14 OMP: found 9592 primes; Time:         5 msec (threads=102)
15 TBB: found 9592 primes; Time:      105 msec
16 ==== Run reducepolicy on 128 threads ====
17 OMP: found 9592 primes; Time:         4 msec (threads=128)
18 TBB: found 9592 primes; Time:       54 msec
```



# Reductions



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.

```
1 // range.cxx
2 #   pragma omp parallel for reduction(+:count)
3     for ( auto e : data )
4         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])  
6  for (int it=0; it<nthreads; it++) {  
7      for (int i=0; i<nthreads; i++)  
8          datadata[i]++;  
9  }
```



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

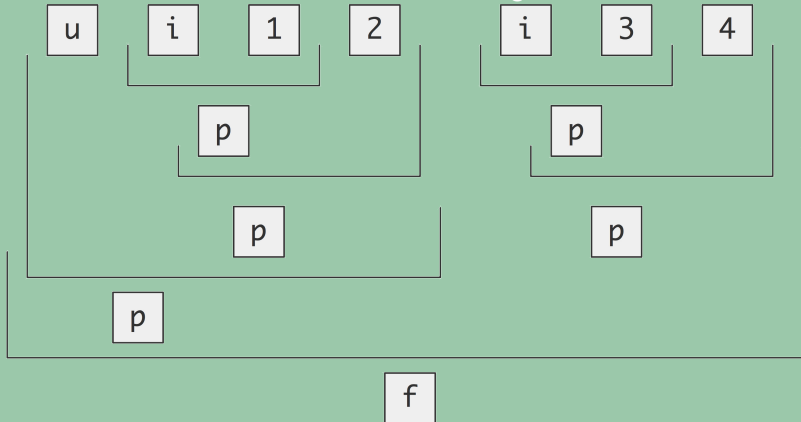
```
1 // reductclass.cxx
2 class Thing {
3 private:
4     float x{0.f};
5 public:
6     Thing() = default;
7     Thing( float x ) : x(x) {};
8     Thing operator+
9         ( const Thing& other ) {
10         return Thing( x + other.x );
11     };
12 };
```

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

A default constructor is required for the internally used init value; see figure 32.



Reduction of four items on two threads, taking into account initial values.





```
1  #pragma omp declare reduction  
2  ( identifier : typelist : combiner )  
3  [initializer(initializer-expression)]  
4
```



## 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:

```
1 // reductexpr.cxx
2 #pragma omp declare reduction\
3   (minabs : int : \
4     omp_out = \
5       [] (int x,int y) -> int { \
6         return abs(x) > abs(y) ? abs(y) : abs(x); } \
7     (omp_in,omp_out) ) \
8   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.



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



Declare a reduction on a histogram object:

```
1  using CharCounter = bincounter<char>;
2  #pragma omp declare reduction\
3  (      \
4          +:CharCounter:omp_out += omp_in \
5          ) \
6  initializer( omp_priv = CharCounter{} )
7  CharCounter charcount;
8  #pragma omp parallel for reduction(+ : charcount)
9  for ( int i=0; i<text.size(); i++ )
10     charcount.inc( text[i] );
```

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



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

```
1 // charcount.cxx
2 template<typename key>
3 class bincounter : public map<key,int> {
4 public:
5 // merge this with other map
6 void operator+=( const bincounter<key>& other ) {
7     for ( auto [k,v] : other )
8         if ( map<key,int>::contains(k) ) // c++20
9             this->at(k) += v;
10        else
11            this->insert( {k,v} );
12 };
13 // insert one char in this map
14 void inc(char k) {
15     if ( map<key,int>::contains(k) )
16         this->at(k) += 1;
17     else
18         this->insert( {k,1} );
19 };
20 };
```



Use atomics because there is no reduction mechanism:

```
1 // mapreduceatomic.cxx
2 class CharCounter : public array<atomic<int>,26> {
3 public:
4     CharCounter() {
5         for ( int ic=0; ic<26; ic++ )
6             (*this)[ic] = 0;
7     };
8 // insert one char in this map
9     void inc(char k) {
10         if (k==' ') return;
11         int ik = k-'a';
12         (*this)[ik]++;
13     };
14 };
```



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`



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  };
```



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
```



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
```



The task spins until it's its turn:

Code:

```
1  # pragma omp task \  
2      shared(filtered,ithread)  
3      {  
4  // wait your turn  
5      while (threadnum>ithread) {  
6  #      pragma omp taskyield  
7      }  
8  // merge  
9      filtered += local;  
10     ithread++;  
11 }
```

Output:

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



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





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:

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

However, the following works:

```
1  // privaterandom.cxx  
2  static random_device rd;  
3  static mt19937 rng;  
4  #pragma omp threadprivate(rd)  
5  #pragma omp threadprivate(rng)  
6  
7  int main() {  
8  
9  #pragma omp parallel  
10     rng = mt19937(rd());
```



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

```
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.



Trick to create a vector of uninitialized data:

```
1 // heatalloc.cxx
2 template<typename T>
3 struct uninitialized {
4     uninitialized() {};
5     T val;
6     constexpr operator T() const {return val;};
7     T operator=( const T&& v ) { val = v; return val; };
8 };
```

so that we can create vectors that behave normally:

```
1 vector<uninitialized<double>> x(N),y(N);
2
3 #pragma omp parallel for
4 for (int i=0; i<N; i++)
5     y[i] = x[i] = 0.;
6 x[0] = 0; x[N-1] = 1.;
```

(Question: why not use `reserve?`)



Easy way of dealing with this:

```
1  template<typename T>
2  class ompvector : public vector<uninitialized<T>> {
3  public:
4      ompvector( size_t s )
5          : vector<uninitialized<T>>::vector<uninitialized<T>>(s) {};
6  };
```



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?)



```
1 // lockobject.cxx
2 class atomic_int {
3 private:
4     omp_lock_t the_lock;
5     int _value{0};
6 public:
7     atomic_int() {
8         omp_init_lock(&the_lock);
9     };
10    atomic_int( const atomic_int& )
11        = delete;
12    atomic_int& operator=( const atomic_int& )
13        = delete;
14    ~atomic_int() {
15        omp_destroy_lock(&the_lock);
16    };
```



```
1  int operator +=( int i ) {  
2  // atomic increment  
3      omp_set_lock(&the_lock);  
4      _value += i; int rv = _value;  
5      omp_unset_lock(&the_lock);  
6      return rv;  
7  };
```





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

Native C++ atomics:

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

Native solution is 10x faster.



```
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



Does not compile:

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

More subtle:

Code:

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

Output:

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

