TACC

Using OpenMP from C++

Victor Eijkhout

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.

```
1  // hello.cxx
2  #pragma omp parallel
3    {
4      int t = omp_get_thread_num();
5      stringstream proctext;
6      proctext << "Hello world from " << t << endl;
7      cerr << proctext.str();
8    }
```


Parallel regions in lambdas

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 for class methods

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

Code:

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

Output:

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

Privatizing class members

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

```
      class foo {
        private:
        int x;
        public:
        void f() {
#pragma omp parallel private x
g()
        }
      }

```

So `f` can not be `static`, and

```
      class foo { public: int x; }
      foo x;
      #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();
  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

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 policies on algorithms.

Range syntax

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.

General idea

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

C++ ranges header

The C++20 ranges library is also supported:

```
1  // range.cxx
2  #     pragma omp parallel for reduction(+:count)
3        for ( auto e : data )
4          count += e;
5  #     pragma omp parallel for reduction(+:count)
6        for ( auto e : data
7                | std::ranges::views::drop(1) )
8          count += e;
9  #     pragma omp parallel for reduction(+:count)
10       for ( auto e : data
11               | std::ranges::views::transform
12               ( []( auto e ) { return 2*e; } ) )
13         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
```


Custom iterators, 0

Recall that

Short hand:

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

for:

```
for ( vector<float>::iterator
        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 ++.

Custom iterators, 1

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 << " ";
```

Custom iterators, 2

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-( const NewVector::iter& other ) const;
7  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.

```
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); };
```

Custom iterators, solution

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

Custom iterators, solution

```
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-( const NewVector<T>::iter& other
       ↪) const {
7    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

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
```

Reductions vs standard parallelism

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

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.

Reductions on vectors

Use the `data` method to extract the array on which to reduce. Also, 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 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.

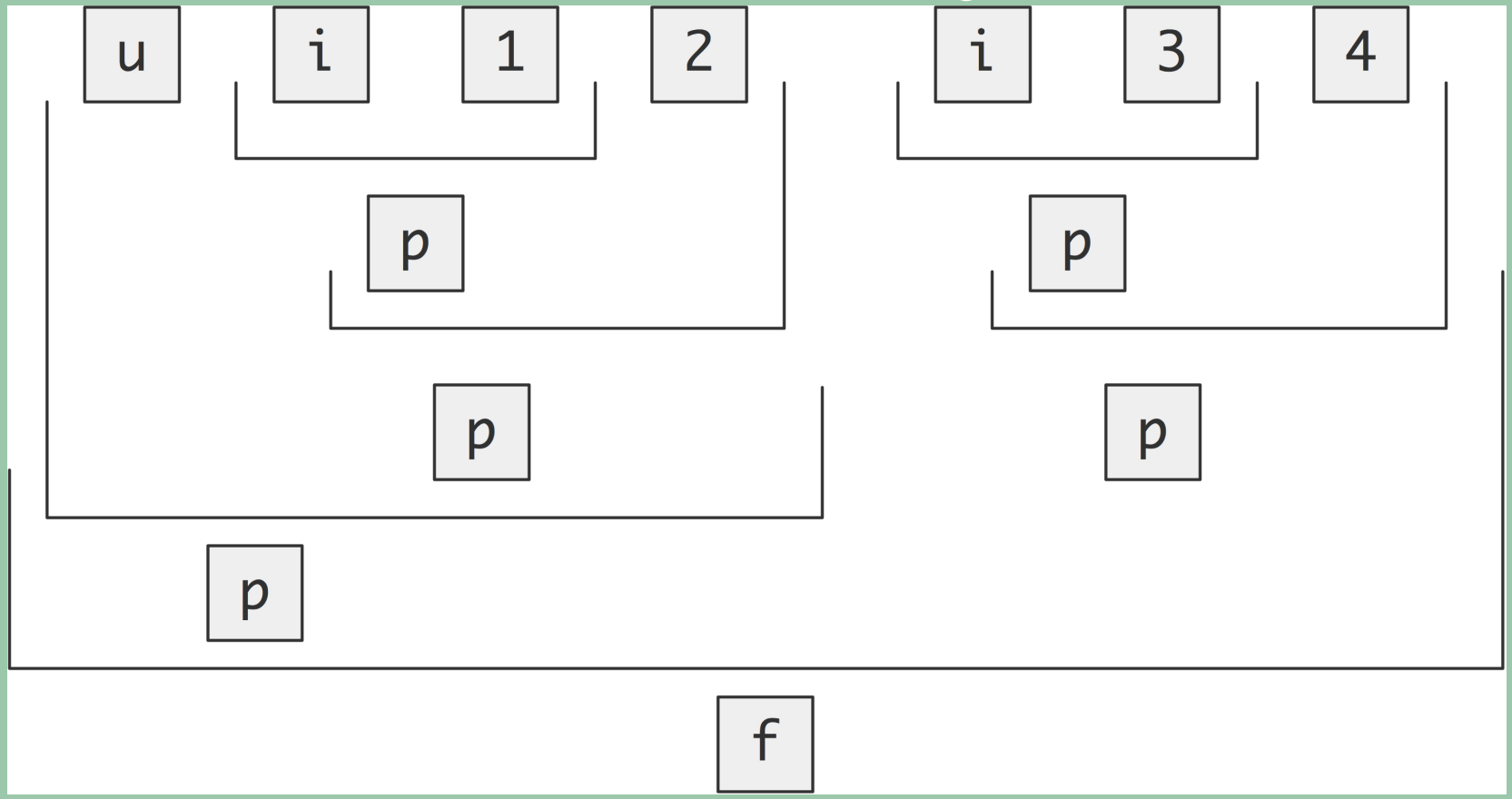
```
// reductcomplex.cxx
class Thing {
private:
  float x;
public:
  Thing() : Thing( 0.f ) {};
  Thing( float x ) : x(x) {};
  Thing operator+( const Thing&
    ↪other ) {
    return Thing( x + other.x );
  };
};
```

```
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 31.

Reduction illustrated

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



User-defined reductions, syntax

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


Reduction over iterators

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(), omp_in.end()) //
       ↪combiner
5      )
```


Lambda expressions in declared reductions

TACC

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.

Example category: histograms

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

Histogram: intended main program

Declare a reduction on a history gram object:

```
using CharCounter = bincounter<char>;
#pragma omp declare reduction\
(       \
        +:CharCounter:omp_out += omp_in \
        ) \
initializer( omp_priv = CharCounter{} )
CharCounter 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: reduction operator

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

```
1  // mapreduce.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 };
```

Histogram in native C++

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 };
```

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
```

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

With a task it becomes possible to have a spin-wait loop:

Code:

```
// filtertask.cxx
#   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

TACC

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

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());
```

You can then use the generator safely and independently:

```
#pragma omp parallel
```


Uninitialized containers

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.

Uninitialized containers, 2

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*?)

Uninitialized containers, 3

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  };
```


Atomic updates

Pragma *atomic* only works for simple cases. Can you atomically do more complicated updates?

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

Atomic updates: class with OMP lock

```
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   };
```

Atomic updates: atomic ops

```
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  };
```


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:

```
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;
4        update<NTHREADS*nops;
5        update++) {
6    my_object += 1;
7  }
8  result = my_object;
```

Native solution is 10x faster.

False sharing prevention

```
#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;

```

Beware vector-of-bool!

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 );
```

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.

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 . )
```