Using OpenMP from C++
Victor Eijkhout
TACC training 2023

Justification

OpenMP has the opportunity to exploit features of modern C++ that are no
present in C. In this course we will explore:

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

Victor Eijkhout

Basic stuff

Victor Eijkhout

1. 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.

```cpp
// hello.cxx
#pragma omp parallel
  {
    int t = omp_get_thread_num();
    stringstream proctext;
    proctext << "Hello world from " << t << endl;
    cerr << proctext.str();
  }
```

Victor Eijkhout

2. 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')

3. Dynamic scope for class methods

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

Code:

```
// nested.cxx
class c {
public:
  void f() {
    cout
      << omp_get_num_threads()
      << '\n';
  };
};
int main() {
  c 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
```

4. Privatizing class members

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

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

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

```
1      class foo { public: int x; }
2      foo x;
3      #pragma omp parallel private thing.x // NOPE
4
```

Victor Eijkhout

5. 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,
```

Victor Eijkhout

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,
```

Victor Eijkhout

Parallel loops

Victor Eijkhout

7. Range syntax

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

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

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

8. C++ ranges header

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

```cpp
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;
```

Victor Eijkhout

9. C++ ranges speedup

```
1  ==== Run range on 1 threads ====
2  Executing: OMP_PROC_BIND=true OMP_NUM_THREADS=1 ./range
3  sum of vector: 50000005000000 in 6.441
4  sum w/ drop 1: 50000004999999 in 6.584
5  sum times 2  : 100000010000000 in 6.896
6  ==== Run range on 2 threads ====
7  Executing: OMP_PROC_BIND=true OMP_NUM_THREADS=2 ./range
8  sum of vector: 50000005000000 in 3.351
9  sum w/ drop 1: 50000004999999 in 3.052
10 sum times 2  : 100000010000000 in 3.205
11 ==== Run range on 8 threads ====
12 Executing: OMP_PROC_BIND=true OMP_NUM_THREADS=8 ./range
13 sum of vector: 50000005000000 in 2.843
14 sum w/ drop 1: 50000004999999 in 1.607
15 sum times 2  : 100000010000000 in 1.438
16 ==== Run range on 40 threads ====
17 Executing: OMP_PROC_BIND=true OMP_NUM_THREADS=40 ./range
18 sum of vector: 50000005000000 in 2.366
19 sum w/ drop 1: 50000004999999 in 0.316
20 sum times 2  : 100000010000000 in 0.308
21   scaling results in: range-scaling-ls6.out
```

Victor Eijkhout

10. Custom iterators, 0

Recall that

Short hand:

```
1        vector<float> v;
2        for ( auto e : v )
3         ... e ...
4
```

for:

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

If we want

```
1        for ( auto e : my_object )
2        ... e ...
3
```

we need an iterator class with methods such as * and ++.

Victor Eijkhout

11. Custom iterators, 1

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

Class:

```
1  // iterator.cxx
2  class NewVector {
3  public:
4  // iterator stuff
5    class iter;
6    iter begin();
7    iter end();
8
9  };
```

Main:

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

12. Custom iterators, 2

Required iterator methods:

```
1  NewVector::iter& operator++();
2  int& 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::iter& operator+=( int add );
```

13. Custom iterators, exercise

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

```
1 class NewVector::iter {
2 private: int *searcher;
3 NewVector::iter::iter( int *searcher )
4   : searcher(searcher) {};
5 NewVector::iter NewVector::begin() {
6   return NewVector::iter(storage); };
7 NewVector::iter NewVector::end()   {
8   return NewVector::iter(storage+NewVector::s); };
```

Victor Eijkhout

14. Custom iterators, solution

```
1  NewVector::iter& NewVector::iter::operator++() {
2    searcher++; return *this; };
3  int&  NewVector::iter::operator*() {
4    return *searcher; };
5  bool  NewVector::iter::operator==( const NewVector::iter &other ) const {
6    return searcher==other.searcher; };
7  bool  NewVector::iter::operator!=( const NewVector::iter &other ) const {
8    return searcher!=other.searcher; };
9  // needed to OpenMP
10 int    NewVector::iter::operator-( const NewVector::iter& other ) const {
11   return searcher-other.searcher; };
12 NewVector::iter& NewVector::iter::operator+=( int add ) {
13   searcher += add; return *this; };
```

Victor Eijkhout

15. OpenMP vs standard parallelism

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

Victor Eijkhout

16. Timing

```
1  OMP: Time:      8591 msec (threads= 5)
2  TBB: Time:      8335 msec
3  OMP: Time:      4298 msec (threads=10)
4  TBB: Time:      4160 msec
5  OMP: Time:      2150 msec (threads=20)
6  TBB: Time:      2082 msec
7  OMP: Time:      1078 msec (threads=40)
8  TBB: Time:      1138 msec
9  OMP: Time:       771 msec (threads=56)
10 TBB: Time:       885 msec
```

TBB slights better on one socket, worse on two.

Reductions

Victor Eijkhout

17. Reductions on vectors

Use the `data` method to extract the array on which to reduce. Also, the reduc
clause wants a variable, not an expression, for the array, so you need an extra
pointer:

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

18. Reduction on class objects

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

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

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

A default constructor is required for the internally used init value; see figure 2

Victor Eijkhout

19. Reduction illustrated

Reduction of four items on two threads, taking into account initial values.
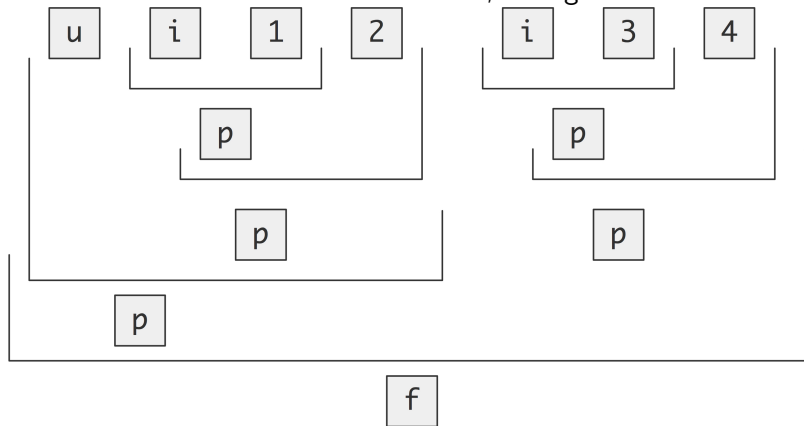
20. User-defined reductions, syntax

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

21. Reduction over iterators

Support for *C++ iterators*

```
1  #pragma omp declare reduction (merge : std::vector<int>
2      : omp_out.insert(omp_out.end(), omp_in.begin(), omp_in.end()))
```

Victor Eijkhout

22. Lambda expressions in declared reductions

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.

Victor Eijkhout

23. Example category: histograms

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

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

24. Histogram: intended main program

```cpp
1  using CharCounter = bincounter<char>;
2  #pragma omp declare reduction\
3  (          \
4             +:CharCounter:omp_out += omp_in \
5             ) \
6  initializer( omp_priv = CharCounter{} )
7
8  string text{"the quick brown fox jumps over the lazy dog"};
9  CharCounter charcount;
10 #pragma omp parallel for reduction(+ : charcount)
11 for ( int i=0; i<text.size(); i++ )
12   charcount.inc( text[i] );
```

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

25. Histogram: reduction operator

```cpp
// mapreduce.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) ) // c++20
        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} );
  };
};
```

Victor Eijkhout

26. Example category: list filtering

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  }
```

27. List filtering, solution 1

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  };
```

Victor Eijkhout

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
        ↪10 15 20 25 30
        ↪40 45 50 55 60
        ↪70 75
```

Victor Eijkhout

29. 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 3
     ↪35 40 45 50 55
     ↪65 70 75 80 85
     ↪95 100
```

30. Templated reductions

You can reduce with a templated function if you put both the declaration and
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);
```

Victor Eijkhout

More topics

Victor Eijkhout

31. 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:

```
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());
```

You can then use the generator safely and independently:

```
1  #pragma omp parallel
```

32. 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.

Victor Eijkhout

33. 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.;
```

Victor Eijkhout

34. 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.

Victor Eijkhout

35. 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    };
```

Victor Eijkhout

```
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  };
```

Victor Eijkhout

```
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();
```

Victor Eijkhout

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.

Victor Eijkhout

39. False sharing prevention

```
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
```

Victor Eijkhout

40. 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: 100000
2  #threads=2; should be
     ↪million: 100000
3  #threads=3; should be
     ↪million: 999964
4  #threads=4; should be
     ↪million: 999659
```

Different `bits[i]` are falsely shared.

Victor Eijkhout

41. CMake

```cmake
cmake_minimum_required( VERSION 3.12 )
project( ${PROJECT_NAME} VERSION 1.0 )

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

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

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