C++ and OpenMP

Christian Terboven

terboven@rz.rwth-aachen.de

Center for Computing and Communication
RWTH Aachen University, Germany

Agenda

- OpenMP and object-oriented programming
- Thread Safety and the STL
- OpenMP and C++ libraries
- Conclusion

Agenda

- OpenMP and object-oriented programming
  - Scoping variables of class-type
  - Parallelization of non-conforming loops
  - Remark on Parallelization of oo-codes
- Thread Safety
- OpenMP and C++ libraries
- Conclusion

Scoping variables of class-type

Simple class for demonstration purposes:

```
class Object1 {
 public:
    Object1();                                   // constr.
    ~Object1();                                  // destr.
    Object1(const Object1& o);                   // copy constr.
    Object1 & operator=(const Object1& o);  // assignm. op.
};
```

- What happens, if instances of such an object are scoped in a parallel region using the different scoping attributes

  (1) as *shared* (2) as *private*

  (3) as *firstprivate* (4) as *lastprivate*

- What happens, if instances of such an object are declared

  (5) as *threadprivate* (5) as *threadprivate + copyin*

Scoping variables of class-type

Let's assume we have declared an instance of Object1

```
Object1 o;
```

and it is *shared* in a parallel region:

```
#pragma omp parallel shared(o)
{ ... }
```

- Simplified excerpt of the C++ standard:
  - The lifetime of an object begins when appropriate storage is obtained and the constructor call (if not non-trivial) has completed
  - Thus, the object's lifetime begins sometime before the parallel region, and ends sometime after it
- OpenMP specification:
  - *Shared variable*: a variable whose name provides access to the same block of storage for all threads in a team
- Conclusion for *shared*:
  - Only the C++ rules for object lifetime apply

Scoping variables of class-type

- What if the variable is *privatized* in a parallel region:

```
#pragma omp parallel private(o)
```

- OpenMP specification:
  - *Private variable*: a variable whose name provides access to a different block of storage for all threads
  - *Private clause*: a new list item of the same type, with automatic storage duration, is allocated for the construct
- Simplified excerpt of the C++ standard:
  - The storage for these objects lasts until the block in which they are created exits
- Conclusion for *private*:
  - Each thread has it‘s own instance of the object, the default constructor is called
  - At the end of the parallel region, the destructor is called
  - The order of constructor calls and destructor calls is undefined

Scoping variables of class-type

- What about *firstprivate* and *lastprivate* variables:

```
#pragma omp parallel do firstprivate(o) / lastprivate(o)
```

- OpenMP specification:
  - *Firstprivate clause*: ... list items private to a thread, initializes each of them with the value that the corresponding original item has ...
  - C/C++: For class types, a copy constructor is invoked to perform the initialization, the order in which copy constructors for different objects are called is unspecified
  - *Lastprivate clause*: ... list items private to a thread, and causes the corresponding original list item to be updated after the end of the region
  - C/C++: For class types, a copy assignment operator is invoked to perform the operation, the order is unspecified again

Scoping variables of class-type

- Conclusion for *firstprivate* and *lastprivate*:
  - Each thread has it's own instance of the object, a copy constructor is called for initialization with *firstprivate*, a copy assignment operator is called to save the value back with *lastprivate*
  - The functions have to be declared conforming and accessible

- What about *threadprivate* variables:

```
#pragma omp threadprivate(o)
```

- OpenMP specification:
  - *Threadprivate directive*: ... specifies that named global-lifetime objects are replicated, each thread has it's own copy
- Conclusion for *threadprivate*:
  - The constructor is called sometime before the first access to the object, the destructor is called sometime after the last access to the object

Scoping variables of class-type

- Last but not least: *threadprivate* + *copyin*, OpenMP specification:
  - The copy assignment operator is invoked

- Now, do the compilers behave as explained?
  - All compilers do fine for `shared`
  - Most compilers do fine for `private`, `firstprivate`, `lastprivate`
    - Some fail: objects are neither constructed nor initialized
  - The tested compilers differ in how they handle `threadprivate` and `threadprivate` with `copyin` / `copyprivate`
    - Objects are not initialized
    - Objects are not destructed
- Proposed workaround:
  - Use private pointers instead of object types, construct and destruct objects using these pointers inside the parallel region

RWTH Center for Computing and Communication

OpenMP and classes

- What is missing in the OpenMP specification:
  - Privatization of (static) class member variables is not possible

- What is bothering in the OpenMP specification:
  - Loop index variables must be of signed integer type, therefore `size_t` is not allowed (depending on the compiler no error is thrown, but parallel region is serialized)

- What you have to care about:
  - If an exception is thrown inside a parallel region, it must be caught inside that parallel region, otherwise the behavior is undefined
  - Using pointers you can get access to everything – but that is not allowed by the OpenMP specification and therefore the behavior is undefined

Parallelization of non-conforming loops

- Parallelization of non-conforming loops:
  - Pointer arithmetic
  - Loops using STL iterators

- Simple example:

```
for (it = list1.begin(); it != list1.end(); it++) {
{
  it->compute();
}
```

- We will now consider three possible solutions …

Parallelization of non-conforming loops

- Construction of a parallelizable loop:

```
long l = 0, lSize = 0;
for (it = list1.begin(); it != list1.end(); it++)
  lSize++;
valarray<CComputeItem*> items(lSize);
for (it = list1.begin(); it != list1.end(); it++) {
  items[l] = &(*it); l++;
}
#pragma omp parallel for default(shared)
for (long l = 0; l < lSize; l++) {
  items[l]->compute();
}
```

Parallelization of non-conforming loops

- Intel‘s Taskqueuing

```
#pragma intel omp parallel taskq
{
for (it = list2.begin(); it != list2.end(); it++) {
#pragma intel omp task
{
   it->compute();
}
} // end for
} // end omp parallel
```

- A similar concept will be available in OpenMP 3.0!

Parallelization of non-conforming loops

- *single-nowait* trick:

```
#pragma omp parallel private(it)
{
for (it = list3.begin(); it != list3.end(); it++) {
#pragma omp single nowait
{
  it->compute();
}
} // end for
} // end omp parallel
```

- Performance of these three techniques
  - depends on the number of loop iterations
  - depends on the amount of work in the loop body
  - depends on the compiler
- Construction of parallelizable loop should be preferred at the moment

Parallelization of oo-codes

- Parallelization of High-Level C++ codes:
  - Internal parallelization
  - External parallelization
  - Thread-safety (next section)
- *Internal parallelization*: complete parallel region inside member functions
  - Main pro argument: no change to the interface
  - Main contra argument: parallel region can not span multice member functions, overhead can not be reduced by enlarging the region
- *External parallelization*: orphaned OpenMP directives inside member functions
  - Main pro argument: parallel region can be enlarged
  - Main contra argument: interface is changed implicitly

Agenda

- OpenMP and object-oriented programming
- Thread Safety and the STL
- OpenMP and C++ libraries
- Conclusion

Thread-Safety

- A function is *reentrant*, if
  - it only uses variables from the stack
  - it only depends on its actual arguments
  - and all its callees fulfill these claims

- A code is *thread-safe*, if it behaves *correct* when run with or called by multiple threads

- Current STL implementations claim to be thread-safe, but what does that mean?   Examination of:
  - Sun C++ libCstd
  - Sun C++ stlport4
  - GNU C++ STL since gcc 3.4
  - Intel C++ since 8.1 (partly building on gcc‘s STL)

Thread-Safety

- Two scenarios:
  - Multiple threads accessing one instance of an STL datatype
  - Multiple threads accessing multiples instances of an STL datatype, but not more than one thread access one instance
- As all STL provided functions and operations are reentrant, one can draw the conclusion that:
  - Only read access: safe
  - Multiple threads accessing distinct instances: safe
  - Multiple threads accessing on instance, at least one thread writes: potential race condition. Application is required to implement locking
  - With respect to the universe of different application scenarios, this behavior is probably optimal.
- Sun‘s *libCstd* und *stlport4* contain some allocators with static data (access secured by internal locking)

Agenda

- OpenMP and object-oriented programming
- Thread Safety
- OpenMP and C++ libraries
  - STL: std::valarray and ccNUMA architectures
- Conclusion

RWTH Center for Computing and Communication

std::valarray and NUMA architectures

- Some datatypes are not suited for NUMA architectures because of properties not visible at first sight
- Example: STL datatype `std::valarray`, elements are guaranteed to be initialized with `zero`
- Initialization (first time touching the data) leads to physical memory distribution – or no „distribution“ on NUMA architectures

- Two approaches for optimization:
  - Employment of operating system features (Sun Solaris)
  - Employment of C++ language constructs with OpenMP

- Solaris feature *madvise* with MADV_ACCESS_LWP advice:
  `int madvise(caddr_t addr, size_t len, int advice)`
- Problem: portability

std::valarray and NUMA architectures

- Usage of C++ language features and OpenMP: first-touch initialization of datatypes with same access pattern as in computation

- Three choices:
  - Modification of `std::valarray`: zero-initialization is done by internal methods which can be modified easily
    - Pro: good performance, low effort
    - Con: solution not portable between compilers and plattforms
  - Usage of other datatype (e.g. `std::vector`) which allows for using a custom allocator which can initialize the memory in a distributed fashion
    - Pro: good performance, portable
    - Con: one-time effort for allocator-implementation
  - Usage of other datatype without initialization (e.g. `TNTs Array1D`)
    - Con: multiple modifications in the program code

Agenda

- OpenMP and object-oriented programming
- Thread Safety
- OpenMP and C++ libraries
- Conclusion

RWTH Center for Computing and Communication

Conclusion

- One can combine high level abstractions and performance!
  (not discussed today)

- The combination of OpenMP and C++ works, but the *portability of performance* depends on
  - Platform
  - Operating System
  - Compiler

- There are deficiencies in the current OpenMP specification regarding C++, but they will hopefully be addressed in 3.0.

- The C++ language and programming style might arise special problems – but it also might offer “elegant” solutions.

End

Thank you for your attention.

Questions?

Designing data types

- Can C++ abstraction features be implemented efficiently?

```
la_vector<double> s(_dimension), r(_dimension), ...;
la_matrix_crs<double> A(_rows, _cols, _vals);

while (iter < max_iter && sqrt(sigma) > tol)
{
     p = r + p * beta;    q = s + beta * q;
     x = x + alpha * p;   r = r - q * alpha;

     s = A * r;

     [...]
}
```

- Yes, if done the right way
  - Avoid temporaries and repeated initializations

Designing data types

- Some advises to achieve high performance with C++:
- Pass by reference: always try to pass arguments by reference rather than by value:

```
template<class T> la_vector<T> operator*(la_matrix_crs<T> & lhs,
                                          la_vector<T> & rhs)
{
   // throw exception, if dimensions do not fit
   // efficient implementation of the actual operation
   // …
}
```

- Together with

```
la_vector<T> & la_vector::operator=(la_vector<T> & rhs)
{ ... }
```

  the sparse matrix vector multiplication `s = A * r` does not introduce any overhead over the „plain C" implementation

Designing data types

- Wherever possible initialize variables just once, e.g. at the declaration or by initialization lists:

```
la_vector(size_t stDimension):
       m_stDimension(stDimension), m_vaData((T)0, stDimension)
{ ... }
```

- Make local functions `static`, declare small functions inline:

```
inline const size_t getDimension() const
{ ... }
```

- If there is something `const`, tell the compiler
- Wherever possible, use nameless objects:

```
foo(TMyClass("abc"));
```

  is faster than

```
TMyClass x("abc");
foo(x);
```

  because parameter and object share memory

RWTH Center for Computing and Communication