C++ and OpenMP

Christian Terboven
terboven@rz.rwth-aachen.de

Center for Computing and Communication
RWTH Aachen University, Germany
Agenda

• OpenMP and objects

• OpenMP and generic programming

• Thread Safety and the STL

• Conclusion
Scoping variables of class-type

Simple class for demonstration purposes:

```cpp
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
  (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

```cpp
Object1 o;
```

and it is *shared* in a parallel region:

```cpp
#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
Scoping variables of class-type

• What about \textit{private}: #pragma omp parallel private(o)

• OpenMP specification:
  • \textit{Private variable}: a variable whose name provides access to a different block of storage for all threads
  • \textit{Private clause}: a new list item of the same type, with \texttt{automatic} storage duration, is allocated for the construct

• Simplified excerpt of the C++ standard, on automatic storage:
  • The storage for these objects lasts until the block in which they are created exits

• Conclusion for \textit{private}:
  • Each thread has its 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 \textit{firstprivate} and \textit{lastprivate} variables:
  
  
  
  \begin{verbatim}
  #pragma omp parallel do firstprivate(o) / lastprivate(o)
  \end{verbatim}

• OpenMP specification:
  • \textit{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
  • \textit{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
  • The functions have to be declared conforming and accessible
Scoping variables of class-type

• What about *threadprivate* variables: `#pragma omp threadprivate(o)`

• OpenMP specification:
  - *Threadprivate directive*: ... specifies that named global-lifetime objects are replicated, each thread has its own copy
  - … a threadprivate object is initialized once, in the manner specified by the program …

• We have to differentiate three kinds of initialization:
  - Without initialization: `Object1 o;`
    - Default constructor is called
  - Direct initialization: `Object1 o( (int)23 );`
    - Constructor accepting the argument is called
  - Explicit initialization: `Object1 o = other_instance;`
    - Copy constructor is called
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
  • Wait for next compiler generation officially supporting OpenMP 3.0
Other issues

• What is bothering / missing in the current OpenMP specification:
  • Privatization of (static) class member variables is not possible
    • Will be allowed in OpenMP 3.0
  • 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)
    • Will be allowed in OpenMP 3.0

• 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
Agenda

• OpenMP and objects

• OpenMP and generic programming

• Thread Safety and the STL

• Conclusion
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 … and then look at what OpenMP 3.0 will offer.
Parallelization of non-conforming loops

- Construction of a parallelizable loop:

// save iterators to an array named items
int iSize = list1.size();
valarray<CComputeItem*> items(lSize);
for (it = list1.begin(); it != list1.end(); it++)
{
    items[l] = &(*it); l++;
}

// now run over that array in parallel
#pragma omp parallel for default(shared)
for (int i = 0; i < iSize; i++)
{
    items(l)->compute();
}

// take care of int <-> long and container requirements
Parallelization of non-conforming loops

- Intel's Taskqueuing (currently), or OpenMP 3.0's tasks:

```c++
#pragma intel omp parallel taskq /* omp parallel single */
{
    for (it = list1.begin(); it != list1.end(); it++)
    {
        #pragma intel omp task /* omp task */
        {
            it->compute();
        }
    } // end for
} // end omp parallel

// OpenMP 3.0: see comments
Parallelization of non-conforming loops

• single-nowait trick:

```cpp
#pragma omp parallel private (it)
{
    for (it = list1.begin(); it != list1.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, on the amount of work in the loop body and on the compiler.
Parallelization of non-conforming loops

- OpenMP 3.0 will allow the following for Random Access Iterators:
  
  ```c++
  #pragma omp parallel for
  for (it = list1.begin(); it != list1.end(); it++)
  {
    it->compute();
  }
  ```

- If you need scheduling / chunksize control:
  - Currently you have to rewrite the loop, OpenMP 3.0 will allow parallelization of iterator loops directly
  - If your loop has an unknown number of iterations:
    - Currently you have to use Intel's Taskqueueing, OpenMP 3.0 will have it's own task concept
Agenda

• OpenMP and objects
• OpenMP and generic programming
• OpenMP and the STL
• 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 on top of 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)
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, Linux)
  - 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 platforms
  - 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 objects
• OpenMP and generic programming
• OpenMP and the STL
• Conclusion
Conclusion

• 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 some will be addressed in 3.0. In some cases, you have to program a workaround.

• Some issues will still be left open
  • Don’t try to use your own types in OpenMP reductions
End

Thank you for your attention.

Questions?