Towards a Multicore C++ Standard Template Library

Zalán Szűgyi, Márk Török, Norbert Pataki
Eötvös Loránd University

TÁMOP-4.2.1/B-09/1/KMR-2010-0003

Introduction

- Multicore programming:
  - languages
  - CPUs
  - GPUs
- FastFlow, Intel TBB, OpenMP, Cuda
- Cilk, Cilk++

Introduction (cont.)

- STL is an efficiency bottleneck in multicore programming:
  - Designed for sequential realm
  - Cilk++ does not provide parallel implementation of C++ STL
  - E.g. OpenMP-based algorithms are available
- Reimplementation of STL in Cilk++
  - Containers
  - Algorithms
  - Functors

Cilk++

- C++ extension, based MIT Cilk
- Developed by Intel
- Widely-used multicore language and SDK
- Three keywords added to the language of C++
- Multicore tools in the library
- No parallel STL-like containers, algorithms, etc.

Cilk++ – Fibonacci

- Cilk++:

```
int fib(int n)
{
   int x; int y;
   if (n < 2) return 1;
   else {
            x = cilk_spawn fib(n-1);
            y = fib(n-2);
            cilk_sync;
            return x + y;
   }
}
```

Standard Template Library

- A C++ library of container classes, algorithms, and iterators, functors.
- Generic library, template parameters
- It can be used with any built-in type and with any user-defined type
- Containers and algorithms are independent.
- Based on the generic programming paradigm
- Highly reduced complexity
- Bugless code, standard names

Standard Template Library (cont.)

- Containers: vector, list, map, set, etc.
- Algorithms: sort, count, fill, etc.
- Iterators: bridge the gap

STL – vector

- Class template
- Memory-contiguous container
- RandomAccessIterators
- Automatically resize itself when capacity is full
- Copy elements when reallocates
- Many for loops can be parallelised.
  - Reallocation
  - Copy constructor
  - Filler constructor

STL – vector

- Split the main task as many threads as many cores are in the CPU.
- To solve this problem, we use metaprogram
- Template metaprogramming
  - Runs at compilation time
  - Functional programmming style
  - Based on the template construct of C++

STL – vector

```
template <int n, int corenum>
struct Do_aux
{
   static inline void it(int size)
   {
           const int s = size / corenum;
           cilk_spawn process( n * s, (n+1) * s);
           Do_aux<n-1, corenum>::it(size);
   }
}
```

STL – vector

```
template <int corenum>
struct Do
{
    static inline void it(int size)
    {
            if( size > MIN_GROWSIZE)
            {
                    Do_aux<corenum – 1, corenum>::it(size);
                    cilk_sync;
            }
            else
                    process(0, size);
    }
}
```

STL – vector

- The results: quad core 2.4GHz CPU

size	our solution	std::vector
100000	0.000	0.001
1000000	0.001	0.004
10000000	0.019	0.033
100000000	0.182	0.365

STL – Algorithms

- STL algorithms can be parallelised when RandomAccessIterators are taken
- Overload STL algorithms on RandomAccessIterators (e.g. advance, distance)
- Cilk++ reducer
  - It ensures safe usability
  - After syncronization, copies merge into a single variable
- Reducer properties:
  - Reliable access to nonlocal variables
  - Do not require locks
  - Efficiently
- reducer_opadd: types with associative "+" operator .

STL – Algorithms

```
template <class Iterator, class T>
typename iterator_traits<Iterator>::difference_type
count( Iterator first,
       Iterator last,
       const T& value,
       random_access_iterator_tag)
{
….
}
```

STL – Algorithms

```
template <class Iterator, class T>
typename iterator_traits<Iterator>::difference_type
count( Iterator first,
       Iterator last,
       const T& value )
{
   return count( first, last, value, typename
iterator_traits<Iterator>() );
}
```

STL – Algorithms

- count:

```
cilk::reducer_opadd<
    typename iterator_traits<Iterator>::difference_type> c;
cilk_for(Iterator i = first; i != last; ++i)
{
    if( *i == value)
    {
        ++c;
    }
}
return c.get_value();
```

STL – Algorithms

- The results: quad core 2.4GHz CPU

size	our solution	stl
100000	0.000	0.001
1000000	0.002	0.006
10000000	0.022	0.046
100000000	0.192	0.289

STL – Functors

- STL mainly uses functors to execute user-defined code snippets in the library.
- Functors: predicates to find, orderings, arithmetical and logical operations.
- Defines an operator()
- Functor or Function object can be called as if it is a function
- The code snippets can be inlined if it is defined by functor.

STL – Functors

- Overload algorithm (such as accumulate) on functor’s associativity
- We define functor traits which describes the characteristics of the functor type.
- Similar to iterator traits.
- Functor traits is a template class that consists of typedef
- This class can be specialised for given functor classes

STL – Functors

```
struct __associative{};
struct __non_associative{};

template <class Fun>
struct __functor_traits
{
    typedef __non_associative associativity;
}

template <class T>
struct __functor_traits<plus<T> >
{
    typedef __associative associativity;
};
```

STL – Functors

- Monoid
  - In mathematics:
    - a set of values (types)
    - an associative operation on that set
    - an identity value
    - (integer, +, 0)
    - (real, *, 1)
  - In Cilk++: A type with five functions:
    - reduce(T *left, T *right)
    - identity(T *p)
    - destroy(T *p)
    - allocate(size)
    - deallocate(p)

STL – Functors

```
template <class InputIterator, class T, class Fun>
T accumulate( InputIterator first, InputIterator last, T init, Fun binary_op,
                        random_access_iterator_tag, associativeness )
{
    cilk::reducer<__Monoid<T, Fun> > reducerImp( init );
    cilk_for( ; first != last; ++first )
    {
        reducerImp() = binary_op( reducerImp(), *first );
    }
    return reducerImp();
}
```

STL – Functors

```
template <class InputIterator, class T, class Fun>
T accumulate( InputIterator first, InputIterator last, T init, Fun binary_op)
{
    return accumulate( first, last, init, binary_op, typename
    __functor_traits<Fun>::associativity()
}
```

Conclusion and Future work

- Multicore programming:
  - Interesting and new way of programming
- Cilk++ is a widely-used and beloved extension of C++
- STL is not (yet) prepared for multicore programming.
- STL can be an efficiency bottleneck for multicore environment.
- We reimplemented containers and algorithms.
- Future work: associative containers

Thank you for your attention!

Questions?