Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 279 (3) (2011) 63–72

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Multicore C++ Standard Template Library in a Generative Way

Zalán Szűgyi[1] Márk Török[2] Norbert Pataki[3]

Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest, Hungary

Abstract

Nowadays, the one of the most important challenges in the programming is the efficient usage of multicore processors. Many new programming languages and libraries support multicore programming.
Cilk++ is one of the most well-known languages extension of C++ providing new keywords for multicore programming.
C++ Standard Template Library is efficient generic library but it does not support parallelism. It is optimized to the sequential realm, hence it can be an efficiency bottleneck when it is used in multicore environment.
In this paper we argue for a multicore implementation of C++ Standard Template Library for Cilk++. We consider the implementation of containers, algorithms, and functors as well. Our implementation takes advantage of generative technologies of C++. We also measure the speedup of our implementation.

Keywords: multicore programming, C++, STL, generic programming

1 Introduction

The recent trend to increase core count in processors has led to a renewed interest in the design of both methodologies and mechanisms for the effective parallel programming of shared memory computer architectures. Those methodologies are largely based on traditional approaches of parallel computing.

Usually, low-level approaches supplies the programmers only with primitives for flows-of-control management (creation, destruction), their synchronization and data sharing, which are usually accomplished in critical regions accessed in mutual exclusion (mutex). For instance, POSIX thread library can be used to this purpose. Programming parallel complex applications is this way is certainly hard; tuning

[1] Email: lupin@ludens.elte.hu
[2] Email: tmark@inf.elte.hu
[3] Email: patakino@elte.hu

doi:10.1016/j.entcs.2011.11.038

them for performance is often even harder due to the non-trivial effects induced by memory fences (used to implement mutex) on data replicated in core's caches.

Indeed, memory fences are one of the key sources of performance degradation in communication intensive (e.g. streaming) parallel applications. Avoiding memory fences means not only avoiding locks but also avoiding any kind of atomic operation in memory (e.g. Compare-And-Swap, Fetch-and-Add). While there exists several assessed fence-free solutions for asynchronous symmetric communications, these results cannot be easily extended to asynchronous asymmetric communications, which are necessary to support arbitrary streaming networks.

The important approach to ease programmer's task and improve program efficiency consist in to raise the level of abstraction of concurrency management primitives. For example, threads might be abstracted out in higher-level entities that can be pooled and scheduled in user space possibly according to specific strategies to minimize cache flushing or maximize load balancing of cores. Synchronization primitives can also be abstracted out and associated to semantically meaningful points of the code, such as function calls and returns, loops, etc.

This kind of abstraction significantly simplify the hand-coding of applications. However, it is still too low-level to effectively automatize the optimization of the parallel code: here the most important weakness is in the lack of information concerning the intent of the code (idiom recognition); inter-procedural/component optimization further exacerbates the problem. The generative approach focuses on synthesizing implementations from higher-level specifications rather than transforming them. From this approach, programmers' goal is captured by the specification. In addition, technologies for code generation are well developed (staging, partial evaluation, automatic programming, generative programming) [7]. FastFlow [3], TBB [17] and OpenMP [8] follow this approach. The programmer needs to explicitly define parallel behaviour by using proper constructs, which clearly bound the interactions among flows-of-control, the read-only data, the associativity of accumulation operations, the concurrent access to shared data structures.

Cilk++ is a set of extensions to the C++ programming language that enables multicore programming in the style of MIT Cilk. It offers a quick, easy and reliable way to improve the performance of programs on multicore processors [12].

The *C++ Standard Template Library* (STL) was developed by *generic programming* approach [5]. In this way containers are defined as class templates and many algorithms can be implemented as function templates. Furthermore, algorithms are implemented in a container-independent way, so one can use them with different containers [19]. C++ STL is widely-used because it is a very handy, standard C++ library that contains beneficial containers (like list, vector, map, etc.), a lot of algorithms (like sort, find, count, etc.) among other utilities.

The STL was designed to be extensible. We can add new containers that can work together with the existing algorithms. On the other hand, we can extend the set of algorithms with a new one that can be work together with the existing containers. Iterators bridge the gap between containers and algorithms [15]. The expression problem [24] is solved with this approach. STL also includes adaptor types which transform standard elements of the library for a different functionality [4].

An OpenMP-based multicore implementation of the algorithms of STL is already available [18]. A similar implementation is necessary for the widely-used Cilk++ platform [22]. It would be valuable to extend the parallelism to the containers.

In this paper we argue for a multicored implementation of C++ STL. We examine how a set of containers and algorithms can be developed effectively for the Cilk++ platform with the assistance of generative techniques. We measure the speedup of applications.

This paper is organized as follows. In section 2 the Cilk++ platform is introduced briefly. We present how STL's containers can be implemented with the help of Cilk++ in section 3. New implementation of typical STL algorithms are written in section 4. We present the idea of functor traits and show the advantages of overloaded generic functions on this features in section 5. Finally, this paper concludes in section 6.

2 Cilk++

The Cilk++ language can be used to efficiently execute our application on multicore machines. In this language, applications run in the Cilk++ runtime, which manages parallel execution using computation workers. These workers run on separate Operating System threads and there is one worker per CPU core.

The Cilk++ language is C++ with some additional language features. Parallel work is created when the keyword `cilk_spawn` precedes the invocation of a function. The semantics of spawning differ from a C++ function (or method) call only in that the parent can continue to execute in parallel with the child, instead of waiting for the child to complete as is done in C++. The scheduler in the Cilk++ runtime system takes the responsibility of scheduling the spawned functions on the individual processor cores of the multicore computer. A function cannot safely use the values returned by its children until it executes a `cilk_sync` statement. The `cilk_sync` statement is a local "barrier", not a global one as, for instance, is used in message-passing programming. In addition to explicit synchronization provided by the `cilk_sync` statement, every Cilk function syncs implicitly before it returns, thus ensuring that all of its children terminate before it does. Loops can be parallelized by simply replacing the keyword for with the keyword `cilk_for` keyword, which allows all iterations of the loop to operate in parallel.

Cilk++ is a C++ extension, thus C++ STL can be used as general framework for containers and algorithms. On the other hand, STL is not optimized for multicore environment. In this way, STL can be an efficiency bottleneck as it does not support multicore programming. Furthermore, Cilk++ does not contain a general container and algorithm library.

3 Containers

We reimplemented the vector container of STL to improve its effectiveness. There are several operations on vectors, which can be improved by parallelism, such as creating a large vector and fill its elements with a given value, copying a vector, or growing its internal buffer. Some of these operations are done by the constructor, or copy constructor of vector.

Cilk++ provides high level constructs to support parallelism such as `cilk_for`, however, they do not work inside of constructors. (It is not implemented yet.) Therefore we need to handle the parallelism by lower level constructs: `cilk_spawn`, and `cilk_sync`. The first one starts a new thread and executes a function on it. The second one is waiting for the spawned threads to be terminated.

Working with lower level constructs the programmer has to take care of scheduling manually. In our solution we split the main task to as many threads as many cores are in the CPU. As the number of cores is compile time information a template metaprogram [1] does the splitting process during the compiler compiles the source code. This way we can spare the runtime overhead of determining the number of cores and splitting the process.

The example below shows a skeleton how that template metaprogram splits a compilation process into threads.

```
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 );
    }
};

template <int corenum>
struct Do_aux<0, corenum>
{
    static inline void it( int size )
    {
        const int s = size / corenum;
        process( 0, s );
    }
};

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 );
    }
};
```

The struct `Do` starts the process, and the struct `Do_aux` is its utility class. The struct `Do` has a template argument which stores the number of the cores in the CPU. The number of threads to be created depends on it. The computation process starts by calling `Do`'s static member function called `it`. The argument `size` refers the size

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

Table 1
Comparison by running time

of the vector. If the vector has only few elements, it is not worth to compute it parallel, thus the sequential version of process is called. Otherwise the utility struct `Do_aux` is instantiated and its static member function is invoked to execute the process parallelly.

The technique is based on recursive instantiation of templates. The struct `Do_aux` has two template arguments. The first one is kind of loop variable referring to sub-interval to deal with, and the second is the number of cores. The struct `Do_aux` divides the interval into as many sub-interval as number of cores are in the CPU and invokes the process function to the last sub-interval. After that it instantiates itself with the current loop variable minus one, and the number of cores, and invokes the static member function `it`. When the value of the loop variable is zero then the specialized version of struct `Do_aux` is chosen, and the recursion is stopped. This specialized struct does nothing but invokes the process function to the firs sub-interval.

This splitting process is done by a template metaprogram during the compiler compiles the code. Thus the object code generated by the compiler will be the same as it would generated from the source code below. (Let us suppose there are four cores in CPU.)

```
struct Do
{
    static inline void it( int size )
    {
        if( size > MIN_GROWSIZE )
        {
            const int s = size / 4;
            cilk_spawn process( 3 * s, 4 * s );
            cilk_spawn process( 2 * s, 3 * s );
            cilk_spawn process( s,     2 * s );
            process( 0, s );
            cilk_sync;
        }
        else
            process( 0, size );
    }
};
```

The number of cores, and the constant `MIN_GROWSIZE` are determined by an analyzer program. This program runs before our library is compiled in a computer.

We compared the running time of our solution with the vector of STL. We did this test on a quad core 2.4GHz CPU and did with different size of vectors. The table 1 shows the results in seconds.

Our solution is about twice faster than the vector implementation in STL.

4 Algorithms

Beside containers, algorithms are another widely used part of STL, and the algorithms are also implemented in a sequential way. Thereby we reimplement the algorithms to run parallelly and take advantage of the multicore realm. In this section we introduce the ways, that we applied to reimplement these algorithms in parallel environment.

We can divide the algorithms into different groups. These groups are non-modifying sequence operations, modifying sequence operations, heap, sorting, etc.

To name a few:

- `count`: returns the number of elements in a range that compare equal to a specified value.
- `count_if`: return number of elements in range satisfying condition.
- `find` : returns an iterator to the first element in a range that compares equal to a value, or last if not found.
- `fill` : sets value to all elements in the the specified range.
- `sort` : sorts the elements in the range.

To take advantage of parallelism, the input range of the algorithms must be defined by random access iterators. Otherwise to split the range into smaller ones for the working threads could be much slower than the speed gained by parallelism. To overcome this problem, we overloaded the algorithms with `iterator_tag`s, like the function `advance` is implemented in STL [14]. When the iterator defining a range is random access iterator, the parallel version of algorithm is selected, otherwise the sequential one is chosen.

Seeing that the majority of the algorithms go through the data structure via iterators, this iteration could be changed to `cilk_for` which was proposed by Cilk++. In some cases the access of shared resources takes place inside of the algorithms, which led to development of race condition, this way the insurance of atomicity was our responsibility. We tackled this problem by introducing the reducers that ensures that the given variable modification is atomic.

The example below shows our implementation approach to reimplement an algorithm by `cilk_for`:

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

size	our solution	std::vector
100000	0.000	0.001
1000000	0.002	0.006
10000000	0.022	0.042
100000000	0.193	0.289

Table 2
Average running time of algorithms

```
}
```

The other way to reimplement algorithms is to use `cilk_spawn` and `cilk_sync` terms. In this case we applied the template metaprogram skeleton described in chapter 3.

The table 2 shows the average running time of algorithms. We measured it by different size of input range. The tests are done on a quad core 2.4GHz CPU.

There are an experimental parallel implementation of the algorithms of STL in GCC compiler [18]. This implementation is based on OpenMP technology. That solution is just slightly faster than the serial version of algorithm in our test computer.

5 Functors

Reducers can be used to compute an associative operation on huge amount of pieces of data effectively [9]. Conceptually, a reducer is a variable that can be safely used by multiple threads running in parallel. The runtime system ensures that each worker has access to a private copy of the variable, eliminating the possibility of races without requiring locks. When the threads synchronize, the reducer copies are merged (or “reduced”) into a single variable. The runtime system creates copies only when needed, minimizing overhead.

We argue for a new trait type called *functor traits.* With the assistance of this type it can be described if a functor type implements an associative operation. Algorithms, such as `accumulate`, can be overloaded on this information and execute a more effective version of the algorithm and take advantage of associativeness [10].

Functor traits type is similar to STL’s iterator traits type. Traits consist of typedefs. In the STL generic algorithms can be found that are overloaded on the capability of iterators. For instance, `advance` can run at constant time when work with random access iterators and runs at linear time otherwise.

First, we write two dummy types to describe if a functor type is associative or not:

```
struct __associative{};

struct __non_associative{};
```

The default functor traits is a template class and describes that the general functors are not associative:

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

Typical associative functors from the library set to be associative:

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

template <class T>
struct __functor_traits<multiplies<T> >
{
  typedef __associtive associativity;
};
```

Also, non-standard functors can be defined as associative ones.

Now, we can write the algorithm. We distinguish three different cases, the first one is the functor is not associative, the second one is the functor is associative but the iterators are not random access ones, and third one is the functor is associative and iterators are random access ones:

```
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
        iterator_traits<InputIterator>::iterator_category,
      typename
        __functor_traits<Fun>::associativity() );
}
```

Here we present a general monoid type. It will be used by the specialized algorithm:

```
template <class T, class Fun>
struct __Monoid : cilk::monoid_base<T>
{

  __Monoid( const T& t ) : init( t ) { }

  void identity ( T* p ) const
  {
    new (p) init;
  }

  void reduce ( T* a, T* b) const
  {
    *a = Fun()( *a, *b );
  }

private:
  T init;
};
```

Now, we can write the advanced version of algorithm for associative operations and iterators are random access iterators. The other two cases invoke the usual behaviour of the algorithm.

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

Implementation details about reducers can be found in [9].

6 Conclusion and Future Work

Multicore programming is an interesting new way of programming. Cilk++ is widely-used language, that is an extension of C++. On this platform the STL itself is not prepared for multicore programming and no other container/algorithm library is available in this platform. This way, STL can be an efficiency bottleneck in Cilk++ applications.

In this paper, we argue for a multicore version of the C++ Standard Template Library for Cilk++ platform. We reimplemented containers, as well as, algorithms based on the generic programming paradigm. We have worked out a more advanced framework for functors based on the generative techniques. We have measured the speedup of applications.

We have implemented a set of algorithms and containers. In this paper we have not dealt with associative containers, however, they are also ideal containers for parallelization. One of our major future work is to enhance these containers.

Acknowledgement

The Project is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TÁMOP 4.2.1./B-09/1/KMR-2010-0003).

References

[1] Abrahams, D., Gurtovoy, A.: "C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond", Addison-Wesley (2004).

[2] Aldinucci, M., Danelutto, M., Meneghin, M., Kilpatrick, P., Torquati, M.: *Efficient streaming applications on multi-core with FastFlow: the biosequence alignment test-bed*, in Proc. of Intl. Parallel Computing (PARCO) 2009.

[3] Aldinucci, M., Ruggieri, S., Torquati, M.: *Porting Decision Tree Algorithms to Multicore using FastFlow*, in Proc. of European Conference in Machine Learning and Knowledge Discovery in Databases (ECML PKDD), LNCS **6321**, pp. 7–23.

[4] Alexandrescu, A.: "Modern C++ Design", Addison-Wesley (2001).

[5] Austern, M. H.: "Generic Programming and the STL: Using and Extending the C++ Standard Template Library," Addison-Wesley (1998).

[6] Bishof, H., Gorlatsch, S.: *Generic parallel programming using C++ templates and skeletons*, in Proc. of Domain-Specific Program Generation (International Seminar), 2003, Revised Papers, LNCS **3016**, pp. 107–126.

[7] Czarnecki K., Eisenecker, U. W.: “Generative Programming: Methods, Tools and Applications”, Addison-Wesley (2000).

[8] Dagum, L., Menon, R.: *OpenMP: An industry-standard API for shared-memory programming*, “IEEE Computational Science and Engineering” **5** (1998), pp. 46–55.

[9] Frigo, M., Halpern, P., Leiserson, C. E., Lewin-Berlin, S.: *Reducers and other cilk++ hyperobjects*, In Proc. of Symposium on Parallel Algorithms and Architectures (SPAA) 2009, pp. 79–90.

[10] Gottschling, P., Lumsdaine, A.: *Integrating semantics and compilation: using c++ concepts to develop robust and efficient reusable libraries*, in Proc. of the 7th international conference on Generative programming and component engineering, GPCE 2008, pp. 67–76.

[11] Harrison, N., Meiners, J. H.: *The dynamics of changing dynamic memory allocation in a large-scale C++ application*, In Companion To the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (Portland, Oregon, USA, October 22 - 26, 2006) (OOPSLA, 2006), pp. 866–873.

[12] Leiserson, C. E.: *The Cilk++ Concurrency Platform*, in Proc. of 46th Annual Design Automation Conference 2009, pp. 522–527.

[13] Matsuda, M., Sato, M., Ishikawa, Y.: *Parallel Array Class Implementation Using C++ STL Adaptors*, In Proc. of the Scientific Computing in Object-Oriented Parallel Environments, LNCS **1343**, pp. 113–120.

[14] Meyers, S.: “Effective STL – 50 Specific Ways to Improve Your Use of the Standard Template Library”, Addison-Wesley(2001).

[15] Pataki, N., Szűgyi, Z., Dévai, G.: *C++ Standard Template Library in a Safer Way*, In Proc. of Workshop on Generative Technologies 2010 (WGT 2010), pp. 46–55.

[16] Pataki, N., Szűgyi, Z., Dévai, G.: *Measuring the Overhead of C++ Standard Template Library Safe Variants*, “Electronic Notes in Theoretical Computer Science” **264(5)**, pp. 71–83.

[17] Reinders, J.: “Intel Threading Building Blocks”, O’Reilly (2007).

[18] Singler, J., Sanders, P., Putze, F.: *The Multi-Core Standard Template Library*, In Proc. of 13th International Euro-Par Conference, LNCS **4641**, pp. 682–694.

[19] Stroustrup, B.: “The C++ Programming Language”, Special Edition, Addison-Wesley, 2000.

[20] Szűgyi, Z., Pataki, N.: *Sophisticated Methods in C++*, In Proc. of International Scientific Conference on Computer Science and Engineering (CSE 2010), pp. 93–100.

[21] Szűgyi, Z., Sinkovics, Á., Pataki, N., Porkoláb, Z.: *C++ Metastring Library and its Applications*, In Proc. of Generative and Transformational Techniques in Software Engineering 2009, LNCS **6491**, pp. 467–486.

[22] Szűgyi, Z., Török, M., Pataki, N.: *Towards a Multicore C++ Standard Template Library*, in Proc. of Workshop on Generative Technologies (WGT 2011), pp. 38–48.

[23] Szűgyi, Z., Török, M., Pataki, N., Kozsik, T.: *Multicore C++ Standard Template Library with C++0x*, in AIP Conf. Proc. Vol. **1389**, Numerical Analysis and Applied Mathematics ICNAAM 2011: International Conference on Numerical Analysis and Applied Mathematics, pp. 857–860.

[24] Torgersen, M.: *The Expression Problem Revisited – Four New Solutions Using Generics*, in Proc. of European Conference on Object-Oriented Programming (ECOOP) 2004, LNCS **3086**, pp. 123–143.