FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

How to Realize Data-Parallel Algorithmic Skeletons with C++

Jörg Striegnitz

FZJ-ZAM-IB-2002-02

März 2002

(letzte Änderung: 21.03.2002)

Preprint: Proceedings zum 6. Workshop „Parallele Systeme und Algorithmen“, PASA 2002, Karlsruhe, 11.4.2002

How to Realize Data-Parallel Algorithmic Skeletons with C++

Jörg Striegnitz, Central Institute for Applied Mathematics, Research Centre Juelich, Federal Republic of Germany, J.Striegnitz@fz-juelich.de, WWW project page: `http://www.fz-juelich.de/zam/FACT`

Abstract

Many authors have proposed the use of algorithmic skeletons as a high level, machine independent means of developing parallel applications. Until now their implementation and use was restricted to either functional-, or sophisticated imperative languages. In this paper we will show that C++ provides almost all features that are necessary to implement algorithmic skeletons and identify currying as the only one being not an intrinsic of the language. We will demonstrate how to add currying to the language by relying on language-immanent features and how to realize it without having a negative influence on the runtime performance.

1 Introduction

Algorithmic skeletons represent an approach to parallel programming. The basic idea is to replace explicit parallel programming (e.g. using a parallel language, or a message passing library), by the selection and instantiation of a variety of pre-packaged parallel algorithmic forms known as *skeletons* [1, 2].

Usually, skeletons are embedded into a sequential programming language, being the only source of parallelism for a program. Most of them, like for instance *map*, *farm*, and *divide and conquer*, are implemented as polymorphic higher order functions [4]. Since these are features common to functional languages, most languages with skeletons build upon a functional host [1, 3]. There are also some sophisticated imperative languages like SKIL [5] (an extension of C) but until now, there is no implementation of algorithmic skeletons for mainstream-languages like C++.

In [4] three features were identified as key-concepts to enable the implementation of algorithmic skeletons:

- *parametric polymorphism*,
- *higher order functions*,
- *partial application*.

In the following sub-sections we will explain these features in detail and show if- and how they are supported by the C++ language.

1.1 Polymorphic Types

C++ supports parametric polymorphism by means of templates. A template definition is made up of a list of type variables, followed by the definition of a function, a class member function, or a class. For instance, a function template to calculate the average of two values may look as follows:

```
template <typename T>
T average(T a,T b) {
  return (a + b) / 2;
}
```

`T` is a placeholder for any built-in or user-defined C++ type. Substitution with concrete types usually appears to be transparent for the user – he may use `average` as if it were an ordinary C++ function. For instance, if the user passes two integer values to `average`, the C++ compiler automatically generates `int average(int,int)` and different instances of `average` are distinguished by the overloading mechanism of C++.

1.2 Higher Order Functions

Higher order functions are functions that take functions as arguments and/or return functions as result. The implementation of higher order functions is not only possible with C++; the Standard Template Library (STL) [6, 7] – as part of the C++ standard – yet provides lots of examples (e.g. `for_each`, `transform`). Once again templates prove to be a powerful concept and help to express higher order functions in C++. Here is an example:

```
template <typename UNARYOP,typename ARG>
ARG apply_twice(UNARYOP f,ARG x) {
  return f(f(x));
}
```

`UNARYOP` is a placeholder for any C++ type that supports the function call syntax like pointers to functions or classes that provide a parentheses operator – also known as *function call operator*:

```
struct MyFunctionalClass {
  inline int operator()(int a) const {
    return 2 * a;
  }
};
```

An instance of `MyFunctionalClass`[1] behaves like a unary function. Notice the power of this concept: an object may have state and the state could be used in the body of `operator()`. Objects of classes that provide a function call operator are often called *functors* or *functional objects*.

1.3 Partial Application

Partially applying an n-ary function means to bind its first $k < n$ arguments to some fixed values, thereby, yielding an

[1] The keyword `struct` could be used as an equivalent for `class`, except that all members are `public` by default

$n-k$-ary function. The key concept to allow partial application of functions of arbitrary arity is *currying*. Currying has its roots in the mathematical study of functions where it has been shown that it is sufficient to restrict attention to unary functions: every function $f : A_1 \times \cdots \times A_n \rightarrow R$ can be turned into a unary function $g : A_1 \rightarrow \cdots \rightarrow A_n \rightarrow R$, with $\rightarrow$ being associative to the right; g is called the curried form of f. Passing a single argument to g results in a unary function to which we may pass a value of type A_2 to obtain another unary function to which we may pass a value of type A_3 to retrieve yet another unary function to which ... and so on, until we obtain a unary function that takes a value of type A_n and returns a value of type R. Of course $g(a_1) \cdots (a_n)$ is semantically identical to $f(a_1, \cdots, a_n)$. Although g is unary, we may think of it as being n-ary but with the special capability to take its arguments one at a time – calling $g(a_1)...(a_k)$, $k < n$ yields a valid result, namely, a unary function. Semantically, this unary function is the curried form of an $n-k$-ary function h that is defined as follows:

$$h : A_{k+1} \times \cdots \times A_n \rightarrow R$$
$$h(a_{k+1}, \cdots a_n) = f(\underbrace{a_1, \cdots, a_k}_{constant}, a_{k+1} \cdots, a_n)$$

Thus, we retrieve the curried form of f, whereas the first k arguments have been bound to $a_1, \cdots, a_k$.
In C++ support for partial application is limited to binary functions (`bind1st` – see [7]). Moreover, the standard compliant C++ approach appears to be quite complicated. This is due to `bind1st` expecting its first argument to be a functor. Thus, before being able to partially apply an ordinary C++ function, it has to be turned into a functor. This could be done by using the `ptr_fun` adaptor: consider the C++ function `int sum(int a,int b)` which returns `a + b`. Using `bind_1st` and `ptr_fun`, partially applying `sum` to `4` could be done like this:

```
bind1st( ptr_fun(sum) , 4 )
```

the result returned by `bind1st` is a functional object that behaves like a unary function that takes a single integer as argument and returns the value of the argument incremented by four. For instance,

```
bind1st( ptr_fun(sum) , 4 ) (38)
```

will return 42.
We can conclude that C++ offers all the features that are necessary to implement algorithmic skeletons – except for currying of arbitrary functions. In the next section we will show why it is worthwhile to worry about currying and thereafter we will show how currying could be integrated into the language.

2 Data-Parallel Skeletons

Imagine a function `map` which applies a function to every element of a C++ random access container and stores the result in a new container. Obviously, `map` could be implemented as a higher order function:[2]

```
template <typename UNARYOP,
          typename ICONTAINER,
          typename OCONTAINER
         >
OCONTAINER& map(OCONTAINER& o,
                const ICONTAINER& l,
                UNARYOP f) {
  int size = c.end() - c.begin();
  o.clear(); o.reserve(size);
  for (int i=0, i<size, ++i) {
    o[i] = f( c[i] );
  }
  return o;
}
```

The for loop inside of `map` indeed is a very good candidate for parallelization because there are no data dependencies to be worried about. For instance, using OpenMP [16], the parallelization of `map` turns out to be straightforward:

```
template <typename UNARYOP,
          typename ICONTAINER,
          typename OCONTAINER
         >
OCONTAINER& map(OCONTAINER& o,
                const ICONTAINER& l,
                UNARYOP f) {
...
#pragma omp parallel for \
            private(i) schedule(static)
  for (int i=0, i<size, ++i) {
 ...
}
```

It is very important to notice that pursuing other and even much more complicated parallelization strategies (e.g. by using distributed data structures and MPI, or by performing some sort of load-balancing etc.) does not necessarily mean that we have to change the interface of `map` – this is one of the key features of algorithmic skeletons. Another aspect is the flexibility and expressiveness that comes from the combination of the concepts of higher order functions and currying.
Consider for instance a container that stores molecules and that every molecule has to be propagated in space by a certain vector. The following code provides a sketch of a possible solution that relies on `map`:

```
class molecule {
public:
  double x,y;
  ...
};

molecule& move(molecule& m) {
  m.x += 1 ; m.y += 1;
  return m;
}

vector<molecule> md;
map(md,md, move);
```

[2] `UNARYOP` may change the element type that is stored in the C++ container. In that particular case, `ICONTAINER` and `OCONTAINER` are of different type. To keep the examples presented in this paper simple, we leave type computations to the user rather than using template template parameters and template metaprogramming [12] to perform type computations automatically. By using `OCONTAINER` as an input to `map`, the result-type is always clear.

Obviously, pursuing this approach to displace molecules by arbitrary vectors will result in numerous sophisticated functions. We can do better by making use of the observation we made in section 1.3 (classes that encapsulate state and behave like functions):

```
struct mVec {
  int x,y;
  mVec(int _x,int _y) : x(_x),y(_y) {}
  inline
  molecule& operator()(molecule& m) const {
    m.x += x ; m.y += y;
    return m;
  }
};

vector<molecule> md;
map(md,md, mVec(1,4) );
```

here `mVec(1,4)` creates a temporary object of type `mVec` and due to the overloaded function call operator this object behaves like a unary function and hence could be passed to `map`. The `mVec` class allows us to displace molecules by arbitrary offsets and we still can take benefit from a possible parallel implementation of the `map` skeleton,
However, the drawback of this approach is that defining a class is not as natural as defining a function. If we were able to partially apply functions, we could have implemented the following variant (pseudo-code!):

```
molecule& move(int x,int y,molecule m) {
  m.x += x ; m.y -= y;
  return m;
}

map(md,md, move(2)(3) );
```

`move(2)(3)` would have returned a function that takes a molecule and moves it by the vector (2,3).
With respect to object-oriented design it even would be preferable to make `move` a method of class `molecule`. But we have to take care when currying methods because they take an implicit parameter called `this`. It seems natural to make `this` explicit for the curried version of a method by making it the last parameter to a function. Curried methods then would allow for calling `map(md,md, molecule::move(2)(3))`.
Once you are familiar with currying, it turns out to be quite more natural than providing functional objects that only act as wrappers. Moreover, it enables and supports reuse of existing code.
Unfortunately, although being on Stroustrup's list of possible extensions to C++ [15], C++ has no intrinsic support for partial application – but we can emulate it!

3 Our New Concept: Functional Functors

Our primary goal is to turn a C++ functions into a curried form. Thus, we need to consider how to represent functions in C++; at least three variants are possible:

1. C++ function,
2. functional objects – functors,
3. and our new concept: *functional functors* (in short: `f-functors`)

A C++ function is a function with C++ linkage or a static class member function. Functional objects already have been motivated in section 1.3. f-functors are functors whose function call operator is unary. However, in comparison to usual functors they are more general, because they do not perform computations on their own but delegate computation to a computational rule.
F-functors are implemented as class templates, generalizing argument type(s), return type, and the type of the computational rule. Here is the definition of a functional functor, used to represent curried unary functions:

```
template </* Signature                    */
          typename ARG,typename RESULT,
          /* Computational rule (CR)     */
          typename UNARYOP=RESULT (*)(ARG)>
class FUNC1 {
private:
  const UNARYOP op;
public:
  /* Constructor: needs CR to initialize */
  FUNC1( const UNARYOP &op_ ) : op (op_) {}
  inline RESULT operator()( ARG a ) const {
    /* Hand off argument to CR          */
    return op( a );
  }
};
```

`UNARYOP` is a placeholder for any possible representation of a unary function. In the definition above it defaults to be a pointer to function because we expect this to be the most common situation. Unary functions and their curried forms are identical and hence, nothing special has to be done: `FUNC1`'s function call operator just delegates functionality to the computational rule represented by `op`.
Developing an f-functor to represent binary functions we have to take into account partial application. Consider a binary function f and its curried form g:

$$f : A_1 \times A_2 \to R \; ; \; curry(f) = g : A_1 \to (A_2 \to R)$$

Applying g to a single argument $a_1 \in A_1$ returns a unary function of type $(A_2 \to R)$. In our model unary functions are represented by instances of `FUNC1`. Consequently, an f-functor that represents g should provide a unary parentheses operator that returns an object of type `FUNC1`. Since `FUNC1` is a template, we have to think about the types to use during instantiation. Argument- and result types are dictated by g's signature, but the type of the computational rule still needs to be provided.
The computational rule has to be: take an argument of type A_2 and use it together with the parameter a_1 – that already has been passed to g – and return the result of the application of f to both arguments. Such a computational rule only makes sense in conjunction with `FUNC2` and the functional object representing it (`PAF`) hence, is defined in the local scope of `FUNC2`:

```
template </* Signature for binary function */
          typename ARG1,
```

```
          typename ARG2,
          typename RESULT,
          /* Computational rule (CR)       */
          typename BINOP=RESULT(*)(ARG1,ARG2)
         >
class FUNC2 {
private:
  const BIN op;
public:
  FUNC2( const BIN &op_ ) : op (op_) {}

  /* Partial Application Functor          */
  struct PAF {
    const BINARYOP op; // Memorize operation
    ARG1           a1; // Store argument
                       // that already was
                       // passed to op
    /* Constructor needs CR and argument  */
    PAF(const BINARYOP & op_,ARG1 a1_) :
           op(op_),a1(a1_) {}

    inline
    RESULT operator()(ARG2 a2_) const {
      return op(a1,a2_); // Make use of
                         // memorized
                         // argument
    }
  };

  FUNC1<ARG2,RESULT,PAF>
    operator()( ARG1 a1 ) const {
    return FUNC1<ARG2,RESULT,PAF>(PAF(op,a1));
  }
};
```

At this point we can define an instance of a functional functor to represent the `sum` function and make use of it:

```
/* Note: ommitting type of CR is possible;
   type int (*)(int,int) is assumed
*/
FUNC2<int,int,int> fsum(sum);
result = fsum(1)(2);
```

`fsum(1)(2)` is a valid C++ expression: `fsum(1)` returns an objects of type

```
FUNC1<int,int, FUNC2<int,int, int >::PAF>
```

which indeed offers a unary parentheses operator that expects an integer as argument.

Developing functional functors for functions of arbitrary dimension is straightforward now and we therefore developed a code-generating tool to automate this process. The code generator is supplied with the largest dimension a function to curry may have and produces an appropriate C++ header file.

Usually, defining an f-functor is a seldom task because partial application mostly occurs when a function gets passed to a higher order function. We therefore introduced a function called `curry`. It takes a pointer to a C++ function and returns an appropriate f-functor object. The `curry` function used for binary functions looks like this:

```
template <typename ARG1,typename ARG2,
          typename RESULT>

inline // To avoid a function call to curry
FUNC2<ARG1,ARG2,RESULT, RESULT (*)(ARG1,ARG2)>
  curry(  RESULT (*cfunc)(ARG1,ARG2)  ) {
  return FUNC2<ARG1,ARG2,RESULT,
              RESULT (*)(ARG1,ARG2)>( cfunc );
}
```

Like functional functors, for each dimension a function to curry may have, a suitable curry function is needed. Thus, they are created by the generator tool as well.

With some compilers we have to be a bit careful because the f-functor returned by `curry` gets its computational rule from a pointer to function. This may result in an indirect call when calling the computational rule, thereby, revealing a possible performance penalty due to inlining being forced to stop at this point.

To assist such compilers, we introduced yet another functor type. Like f-functors, it is realized as class template, but now the computational rule occurs as a constant template parameter:

```
template <typename ARG1,typename ARG2,
          typename RESULT,
          /* cfunc is a placeholder for
             a constant, not a type!
          */
          RESULT (* cfunc)(ARG1,ARG2)>
class MakeF2 {
public:
  /* MakeF2 is computational rule of the
     functional functor type it exports.
  */
  typedef FUNC2<ARG1,ARG2,RESULT, MakeF2>
          ffunc_t;

  inline
  RESULT operator()(ARG1 a1,ARG2 a2) const {
    /* Address of 'cfunc' is known during
       compile-time
       -> no penalty causing indirection!
    */
    return cfunc(a1,a2);
  }
};
```

Using `MakeF2` together with `FUNC2` there no longer is an indirection during evaluation. However, the drawback of this approach is that the declaration of an f-functor becomes more complicated (it has to be supplied with a computational rule of type `MakeF2`). For example, a definition of a functional functor for the `sum` function has to have the following form:

```
FUNC2< int,int,int, MakeF2<int,int,int,sum> >
   fsum( MakeF2<int,int,int,sum>() );
```

Fortunately, it is possible to improve on this by introducing yet an additional class template that inherits from `MakeF2`

```
template <typename ARG1,typename ARG2,
          typename RESULT,
          RESULT (* cfunc)(ARG1,ARG2) >
class FUNC2_ptr : public
     MakeF2<ARG1,ARG2,RESULT,cfunc>::ffunc_t {
};
```

Finally, except for the class name, defining a functional functor looks as usual[3].:

```
FUNC2_ptr< int,int,int, sum>  fsum;
```

[3] In the original implementation all f-functors have a private default constructor and the MakeF* classes are declared as their friends. Thus, we do not need a constructor argument at this point

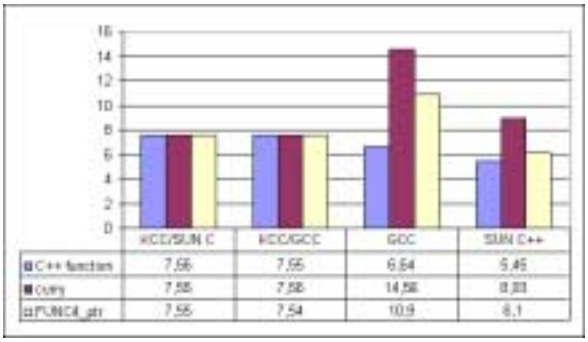

Figure 1 Calling a C++ function and its curried variants. The chart shows the absolute time to execute each kind of evaluation fifty million times. Using a highly optimizing C++ compiler (like Kuck and Associates KCC) there obviously is no negative impact on the performance, if our new concepts get used.

4 Portability and Performance

We have tested our approach on several platforms. So far we were successful with the following C++ compilers:

- KCC 3.4f (Kuck and Associates)
- GCC 2.95.2 (Free Software Foundation)
- Visual C++ 6.0 + Service Pack 3 (Microsoft)
- Workshop Pro C++ 5.0 (Sun)
- Portland Group C++ (Portland Group)
- MipsPro C++ 7.3.1.1 (SGI)

Our testing environment consists of a Sun Ultra10 (UltraSparcIIi with 333 MHz, 384 MB RAM) running under Solaris 7. To estimate the performance of the `curry` function we ran several tests. All of them are based on the following C++ function:

```
inline int sum(int a,int b,int c,int d) {
  return a + b + c + d;
}
```

First we investigated how much time it takes to evaluate this function fifty million times, by either calling it directly, using its curried form obtained by `curry`, or using the curried form obtained through the use of `FUNC4_ptr`.
Since the use of curried variants of a function without partial application is quite unusual, we ran two additional tests. In these tests we tried to reveal the overhead when passing a partially applied function to a higher order function and distinguished whether this higher order function has been declared `inline` or not.
To get an idea on how competitive we are with existing sophisticated languages, we also ran a test using SKIL [5]. SKIL is a sophisticated imperative language – based on C – that has support for polymorphic types, higher order functions and partial application. Since SKIL is based on C we did not distinguish whether the higher order function was declared inline or not. Figure 2 shows that using a highly optimizing C++ compiler we are in fact competitive with SKIL.

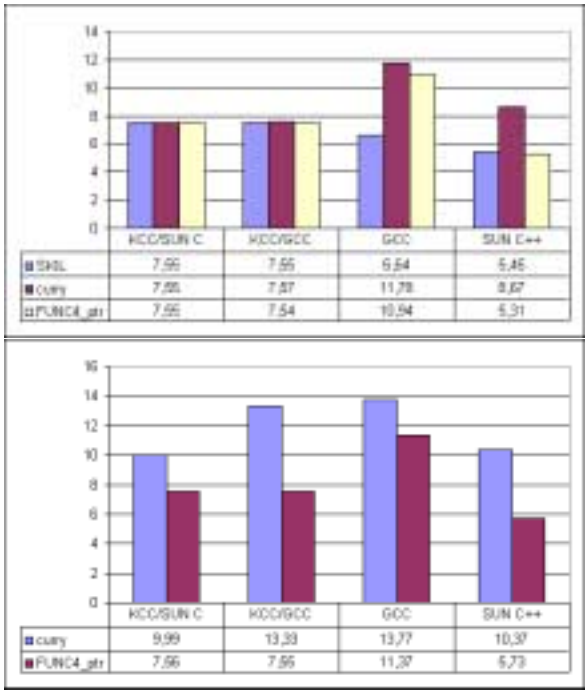

Figure 2 Passing partially applied function to inline function (top) and to non-inline function (bottom).

5 Conclusion and Future Work

Our aim was to figure out whether C++ is a suitable for the implementation of algorithmic skeletons. We identified currying as the only missing feature and demonstrated how it can be integrated into the language be relying on language-immanent concepts.
In summary C++ turns out to be an ideal platform for the implementation of algorithmic skeletons. This even becomes more important as there are upcoming parallel versions of the Standard Template Library (STL) [8, 9] and as there are many higher order functions in the STL that are quite similar to the skeletons proposed in [1] (e.g. `transform` versus `map`).
For further investigations we consider to develop our own parallel version of the STL. Some other research may be to discover whether the programming techniques presented in [10, 11, 12, 17] could be used to implement optimizing code transformations, as proposed in [13, 14].

6 References

[1] J. Darlington, A.J. Field et al.: Parallel Programming Using Skeleton Functions, Proceedings of PARLE '93, LNCS 694, Springer 1993

[2] M. Cole: Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computing, Pitman, 1989.

[3] J. Darlington, Y. Guo et al.: Functional Skeletons for Parallel Coordination, Proceedings of EuroPar '95, LNCS 966, Springer 1995

[4] G.H. Botorog, H. Kuchen: Efficient Parallel Programming with Algorithmic Skeletons, Proceedings of EuroPar '96 Vol. 1, LNCS 1123, Springer 1996
[5] G.H. Botorog, H. Kuchen: Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming, Proceedings of HPDC-5, IEEE Computer Society Press, 1996
[6] International Standard, Programming Languages - C++, ISO/IEC: 14882, 1998
[7] M. Austern: Generic Programming and the STL, Addison Wesley 1999
[8] Silicon Graphics Computer Systems: Parallelizing STL constructs, http://reality.sgi.com/austern/pSTL/parallel-STL.html
[9] E. Johnson, P. Beckman, D. Gannon: HPC++: An Experiment with the Parallel Standard Template Library, http://www.cs.indiana.edu/hyplan/ejohnson/-papers/ics97/ics97.html
[10] T. Veldhuizen, D. Gannon: Active Libraries: Rethinking the roles of compilers and libraries
[11] T. Veldhuizen: Expression Templates, C++ Report Vol. 7 No. 5, June 1995
[12] T. Veldhuizen: Template Meta Programs, C++ Report Vol. 7 No. 4 May 1995
[13] S. Gorlatch: Optimizing Compositions of Scans and Reductions in Parallel Program Derivation, Technical Report MIP-9711, University of Passau, Germany May 1997
[14] S. Gorlatch, S. Pelagatti: A Transformational Framework for Skeletal Programs: Overview and Case Study, Proceedings of IPPS/SPDP '99, Springer 1999
[15] B. Stroustrup: The Design and Evolution of C++, Addison Wesley 1995
[16] Robit Chandra et. al.: Parallel Programming in OpenMP, Morgan Kaufmann 2000
[17] J. Striegnitz, S. Smith: An Expression Template aware Lambda Function, Proceedings of the First Workshop on Template Metaprogramming