Semantic-Driven Parallelization of Loops Operating on User-Defined Containers

Dan Quinlan, Markus Schordan, Qing Yi, and Bronis R. de Supinski

Lawrence Livermore National Laboratory, USA
{dquinlan, schordan1, yi4, bronis}@llnl.gov

Abstract. We describe ROSE, a C++ infrastructure for source-to-source translation, that provides an interface for programmers to easily write their own translators for optimizing user-defined high-level abstractions. Utilizing the semantics of these high-level abstractions, we demonstrate the automatic parallelization of loops that iterate over user-defined containers that have interfaces similar to the lists, vectors and sets in the Standard Template Library (STL). The parallelization is realized in two phases. First, we insert OpenMP directives into a serial program, driven by the recognition of the high-level abstractions, containers, that are thread-safe. Then, we translate the OpenMP directives into library routines that explicitly create and manage parallelism. By providing an interface for the programmer to classify the semantics of their abstractions, we are able to automatically parallelize operations on containers, such as linked-lists, without resorting to complex loop dependence analysis techniques. Our approach is consistent with general goals within *telescoping languages*.

1 Introduction

In object-oriented languages such as C++, abstractions are a key aspect of library design, sharing aspects of language design, which aims to provide the application developer with an efficient and convenient interface. For example, the C++ Standard Template Library (STL), parts of which are standardized within the C++ standard libraries, includes a collection of template classes that can be used as containers for user-defined constructs. Some STL containers, such as vectors, provide random access to their elements using an integer index, while other containers such as lists and sets provide other means to access their elements. Nevertheless, all STL containers provide sequential element accesses and thus all of them can be used in the code fragment in Figure 1. This design strategy permits all containers to be used interchangeably in algorithms that process a sequence of elements.

At this level, library design greatly resembles language design, but without increasing the complexity of the compiler. The term *telescoping languages* was coined by Kennedy [1] in 2000. Within telescoping languages, a base language is chosen and domain-specific types are constructed entirely within the base language with no language extension. The iterative progression of a library to a

higher-level language comes only with compile-time support for its user-defined types. The telescoping aspect relates to the optional use of the compile-time optimizations, because the abstractions are defined fundamentally as a library completely within the base language. The idea of higher-level languages driving the generation of lower-level C++ code was originally discussed by Stroustrup in 1994 [2] (page 204). The techniques presented in this paper are a special case of compiler support for high-level abstractions such as those defined in the STL. Specifically in this paper we utilize the semantics of the high-level abstractions and generate low-level C++ code.

```
MyContainer myContainer;
MyContainer::iterator p;
for (p = myContainer.begin(); p != myContainer.end(); ++p) {
  foo(*p);
}
```

Fig. 1. Example: a code fragment processing a user-defined container

Due to the increasing popularity of the STL library, more and more libraries provide containers that conform to the STL interface. Since the library developer knows the semantics of the library's containers and of each element in the containers, he can write a *source-to-source translator* that optimizes the performance of every program that uses his library. For example, in Figure 1, if the library writer knows that none of the elements in *MyContainer* can be aliased and that the function *foo* is side-effect free (i.e., it does not modify any global variables), he can safely parallelize the surrounding loop and thus achieve better performance for the user's application. Due to the undecidability of precise alias and control-flow analysis, it could be impossible for a compiler to automatically figure out this semantic information. Thus, our approach can better optimize any application code that uses the library since we allow the library developer to communicate this semantic information to the source-to-source translator. The application developer sees only an automated process.

We present ROSE, a C++ source-to-source infrastructure especially for this purpose [3, 4]. In addition to being a general source-to-source compiler infrastructure, ROSE provides several mechanisms, including a very high level Abstract Syntax Tree (AST) that maintains the original structure of the user program, traversal facilities for modifying the AST, and a string interface for inserting new C++ code fragments (which are represented as strings) into the AST directly. Since we have not only the syntax of the original program but also its full type resolution within the ROSE AST, we can use specific user-defined type information as a basis for optimizing an application. Thus, the compiler has fundamentally more information, enabling greater levels of optimization. In the case of parallelizing user-defined containers, for example, we can automate the introduction of OpenMP directives into otherwise serial code because the library writer guarantees the required semantics. Based on the additional semantics of

user-defined abstractions, this approach permits parallel execution of appropriate fundamentally serial code. Section 2 presents the ROSE infrastructure in more detail.

Using the ROSE approach for processing user-defined abstractions, we present a source-to-source translator that automatically introduces OpenMP directives in loop computations on STL-like container classes such as the one in Figure 1. The only additional information that needs to be provided by the library programmer is the set of container classes that disallow aliased elements and the side-effects of library functions. We then invoke another translator within ROSE to recognize specific OpenMP pragma directives and to translate these directives (along with their associated code fragments). The final result is a parallel program that explicitly creates and manages parallelism.

2 Infrastructure

The ROSE infrastructure offers several components to build a source-to-source translator. A complete C++ frontend is available that generates an object-oriented annotated abstract syntax tree (AST) as an intermediate representation. Several different components can be used to build the midend of a translator that operates on the AST to implement transformations: a predefined traversal mechanism; a restructuring mechanism; and an attribute evaluation mechanism. Other features include parsing of OpenMP directives and integrating these directives into the AST. A C++ backend can be used to unparse the AST and generate C++ code (see Figure 2).

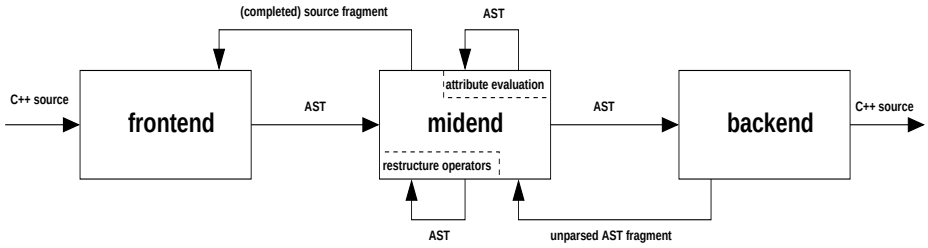

Fig. 2. ROSE Source-To-Source infrastructure with frontend/backend reinvocation

2.1 Frontend

We use the Edison Design Group C++ frontend (EDG) [5] to parse C++ programs. The EDG frontend performs a full type evaluation of the C++ program and then generates an AST, which is represented as a C data structure. We translate this data structure into an object-oriented abstract syntax tree (AST) which is used by the midend as an intermediate representation. We use Sage III

as an intermediate representation, which we have developed as a revision of the Sage II [6] AST restructuring tool.

2.2 Midend

The midend supports restructuring of the AST. The programmer can add code to the AST by specifying a source string using C++ syntax, or by constructing subtrees node by node. A program transformation consists of a series of AST restructuring operations, each of which specifies a location in the AST where a code fragment (specified as a C++ source string or as an AST subtree). should be inserted, deleted, or replaced.

The order of the restructuring operations is based on a pre-defined traversal. A transformation traverses the AST and invokes multiple restructuring operations on the AST. To address the problem of restructuring the AST while traversing it, we make restructuring operations side-effect free functions that define a mapping from one subtree of the AST to another subtree. The new subtree is not inserted until after the complete traversal of the original subtree. We provide interfaces for invoking restructuring operations that buffer these operations to ensure that no subtrees are replaced while they are being traversed.

The midend also provides an attribute evaluation mechanism that allows the computation of arbitrary attribute values for AST nodes. During traversal, context information can be passed down the AST as inherited attributes, and results of transforming a subtree can be passed up the tree as synthesized attributes. Examples for inherited and synthesized attributes include the type information of objects, the sizes of arrays, the nesting levels of loops and the scopes of associated pragma statements. These attributes can then be used to compute constraints on transformations — for example, to decide whether to apply a restructuring operation on a particular AST node.

Our infrastructure supports the use of C++ source strings to define code fragments. Any source string that represents a valid declaration, statement list, or expression can specify a code pattern to be inserted into the AST. The translation of a source code string, s, into an AST fragment, is performed by reinvoking the frontend. Our system extends s to form a complete program, which it then parses into an AST by reinvoking the frontend. From this AST, it finally extracts the AST fragment that corresponds to s. This AST fragment is inserted into the AST of the original program.

Further, we provide an abstract C++ grammar which covers all of C++ and defines the set of all abstract syntax trees. We have integrated an attribute grammar tool which allows the specification of attribute evaluations on the abstract C++ grammar. The grammar is abstract with respect to the concrete C++ grammar and does not contain any C++ syntax. Similar to our traversal mechanism, source-strings and restructure operators can be used in the semantic actions of the attribute grammar. In section 3.3 we show how a transformation can be specified using the abstract grammar, source-strings, and AST restructure operations.

2.3 Backend

The backend unparses the AST and generates C++ source code. It can either unparse all included (header) files or the source file(s) specified on the command line only. This feature is important when transforming user-defined data types, for example, when adding compiler-generated methods. Using this feature preserves all C preprocessor (cpp) control structures (including comments). Output code from the backend appears nearly indistinguishable from input code, except for transformations, to simplify acceptance by users.

The backend can also be invoked during a transformation, to obtain the source code string that corresponds to a subtree of the AST. Such a string can be combined with new code (also represented as a source string) and inserted into the AST.

Both phases, the introduction of OpenMP directives and the translation of OpenMP directives, can be automated using the above mechanisms, as described in the following sections.

3 Parallelizing User-Defined Containers Using OpenMP

Most modern machines have a parallel architecture that requires applications to be efficiently parallelized in order to achieve high performance. The OpenMP standard provides a convenient mechanism to parallelize applications. It extends the traditional sequential languages Fortran, C and C++ to introduce parallelism without requiring the programmer to manage threads or communications explicitly. However, introducing OpenMP directives into a sequential program still requires a significant amount of work, although substantially less than using distributed memory programming models like MPI.

In addition, current use of distributed memory programming models only extends to a subset of the number of processors available on IBM machines at LLNL. Specifically, the limit on the number of MPI tasks requires a hybrid programming model that combines message passing and shared memory programming in order to use all of the machine's processors. These hybrid programming models significantly increase the complexity of the already difficult task of developing scientific applications that include advanced numerical algorithms and physics, and non-trivial geometrics domains. Thus, our approach is particularly useful in extending existing distributed memory applications to use these modern computer architectures effectively. The automated/simplified introduction of parallelism to leverage the shared memory nodes and, thus, a larger part of these machines can significantly improve programmer productivity. The use of dual shared memory and distributed memory programming models is a more general issue within cluster computing (using a connected set of shared memory nodes).

Most C++ programs, including many scientific applications, use high-level abstractions that tailor the user-environment to a specific application domain.

Thus, object-oriented design creates a programming environment that is essentially a programming language that is more domain-specific than a general purpose language could allow, thereby improving programmer productivity.

The ROSE infrastructure provides support for generating source-to-source translators that essentially act as compilers for these domain-specific languages. The designer of the high-level abstractions captures the semantics of those abstractions so that the source-to-source translators can generate high performance code for the user of the domain-specific language. Generally, the designer of the abstractions will be a library writer, although nothing prevents the end user from designing clean interfaces and capturing the semantics for his specific abstractions.

In this section, we present a mechanism to automatically introduce OpenMP directives for user-defined STL-like containers, which is one of the most commonly used user-defined abstractions in object-oriented programming.

3.1 User-Defined Containers

Scientific applications are increasingly using STL, but at present with no path available toward automated shared memory parallelization. Clearly our goal in addressing the optimization (parallelization) of user-defined container classes includes eventually processing STL containers. Such work would have broad impact on how STL could be used within scientific programming.

At present, the ROSE infrastructure does not handle templates sufficiently well to address STL optimization directly. Figure 3 presents a compromise, an example container class that is similar to the STL list class. It has an identical iterator interface, but does not use templates. The example list class accurately reproduces the same iterator interface as is used in STL and more general user-defined containers. The exact details of the iterator interface are not particularly important; our approach could be used to parallelize alternative methods of traversing the elements of containers. Further, the easy construction of compile-time transformations with ROSE could use even more precise semantics of domain-specific containers if necessary.

Figure 4 defines a class to support the automated transformation of iteration on user-defined containers. The automated transformation process introduces new code that uses this supporting class into the application. The `SupportingOmpContainer_list` class builds an array of fixed size, internally, containing pointers to the container's elements. Using this array the class provides indexed access for the OpenMP `parallel for` loop.

3.2 Safety of Parallelization

Our goal is to parallelize loops that iterate over user-defined containers. Given a candidate loop, we must ensure that it is safe to parallelize, that is, dependences must not exist between different iterations of the loop body [7]. Figure 5

```
class list {
  // List class defined similarly to STL List class (but without templates)

  public:
    // fixed element type for list class (to avoid templates)
       typedef int elementType;

  protected:
       struct list_node {
            list_node* next;
            list_node* prev;
            elementType data;
          };

       typedef elementType* pointer;
       typedef elementType& reference;

       typedef list_node* link_type;
       typedef size_t size_type;

  protected:
       link_type first;
       link_type last;

       size_type length;

  public:
       class iterator
          {
            friend class list;
            protected:
                 link_type node;
                 iterator(link_type x);

            public:
                 iterator();
                 bool operator==(const iterator& x) const;
                 bool operator!=(const iterator& x) const;
                 reference operator*() const;
                 iterator& operator++();
                 iterator operator++(int);
          };

       list();

       iterator begin();
       iterator end();

       unsigned int size();
       void push_back(const reference x);
};
```

Fig. 3. Example: Code fragment showing `list` class using iterators.

```
class SupportingOmpContainer_list {
     // This class is used to support the transformation of iterations over STL
     // containers to a form with which we can use OpenMP to parallelize the execution.

     public:
          typedef list::elementType elementType;
          list::elementType** dataPointer;
          unsigned int length;

     public:
          SupportingOmpContainer_list(list & l) {
               length = l.size();
               dataPointer = new list::elementType* [length];
               assert (dataPointer != NULL);

               list::iterator p;
               int i = 0;
               for (p = l.begin(); p != l.end(); p++) {
                    dataPointer[i++] = &(*p);
               }
          }

          unsigned int size() { return length; }
          elementType& operator[](int i) {
             return *dataPointer[i];
          }
};
```

Fig. 4. Example: Code fragment showing the implementation of supporting abstraction for OpenMP translation.

presents our algorithm for this analysis, where `TestParallelLoop` is the top-level function, and function `get_modified_vars` is invoked to compute the set of variables modified by a list of arbitrary statements.

Our algorithm is different from traditional dependence-based approaches in that the library developer supplies domain-specific information to drive the analysis. This information allows us to recognize opportunities of loop parallelization without having to perform aliasing or interprocedural dependence analysis. In Figure 5, this information is represented as the *userSpec* input parameter, which contains the following information from pre-specified inputs by the library developer.

- `known_containers`(*userSpec*) A set of user-defined containers for which the library writer guarantees element uniqueness, i.e., the instances of the container class do not include duplicated elements. All of these containers must have an iterator interface that is similar to Figure 1. Since the elements cannot be aliased to each other, our analysis can safely conclude that it is safe to parallelize a loop that iterates over the container if the loop body does not contain cross-iteration dependences.
- `known_functions`(*userSpec*) A set of user-defined functions whose side effects are known to the library writer. These functions can include both global functions and the member functions of user-defined abstractions.
- `side_effects`(f, *userSpec*) $\forall f \in$ `known_functions`(*userSpec*) The side effects of each function f defined in *userSpec*. Specifically, for each function

```
TestParallelLoop(l, userSpec)
  l: loop to be parallelized;
  userSpec: info. from programmer
  return: whether loop l can be parallelized
header = get_loop_header(l)
body = get_loop_body(l)
if (header iterates over a container c and
    c ∈known_containers(userSpec) )
  cur_elem = get_current_element(c)
  local_vars = get_local_defined_vars(body)
  mod = get_modified_vars(body, userSpec)
  if (mod ==UNKNOWN) return false;
  for (each variable var ∈ mod)
    if (var ∉ local_vars and var ≠ cur_elem)
      return false;
  return true;
return false;
```

```
get_modified_vars(body, userSpec)
   body: statements to be examined;
   userSpec: info. from programmer
   return: variables modified by body
 F = get_function_calls(body);
 modVars = ∅;
 for (each function call f ∈ F)
   if (f ∈known_functions(userSpec) )
     modVars = modVars∪
       side_effect(f, userSpec);
   else return UNKNOWN
 modVars = modVars∪
             get_local_mod_vars(body);
 return modVars
```

Fig. 5. Algorithm for safety analysis of parallelization

$f \in$ `known_functions`($userSpec$), it defines which parameters and global variables can be modified by f. This information allows us to compute the set of variables modified by an arbitrary statement without resorting to inter-procedural side effect analysis.

In Figure 5, the function `get_modified_vars` uses the semantic information of user-defined functions to help determine the side effects at each iteration of the loop body: for each statement within the loop body and for each function invocation f within the statement, if the function does not belong to the known functions in *userSpec*, we assume that the function could induce unknown side effects and thus conservatively disallow the loop parallelization. In addition, the variables locally modified by each statement is also returned as part of the complete side effect of the loop body.

The function `TestParallelLoop` uses both the known containers and known functions from *userSpec* to identify opportunities of loop parallelization. First, we examine the candidate loop to see if it iterates over a container that is known to be safe to be parallelized. We then invoke `get_modified_vars` to summarize the complete side effect of the loop body. To determine the dependence pattern of the loop body, for every variable *var* modified by the loop body, if *var* is exactly the element of the container being accessed by the current iteration, or if *var* is a local variable defined within the loop body, we know that the variable is private to the current iteration and thus cannot introduce cross-iteration dependences; otherwise, we assume that the variable could be aliased to a global variable and disallow the parallelization.

Note that although the algorithm in Figure 5 is more conservative than traditional dependence-based approaches, it provides a way to utilize user-defined semantic information that might not be available to the other systems. For ex-

ample, even the most aggressive parallelizing compilers may not be able to figure out that the elements of a user-defined pointer-container can never be aliased. By configuring our system with general, user-defined type information, we therefore are able to optimize user-defined objects more effectively in various cases.

3.3 OpenMP Transformation

OpenMP transformations are specified as source-to-source translations. The input program is a sequential C++ program in which we introduce OpenMP pragmas and transform parts of the program into a canonical OpenMP form if necessary.

A transformation is specified as semantic actions of an abstract C++ grammar. The grammar is abstract with respect to the concrete C++ grammar and does not include any concrete C++ syntax. The abstract grammar defines the set of all abstract syntax trees (ASTs) and covers full C++. Computations on the abstract grammar can be specified as attribute evaluations. Attributes can be of arbitrary type, including source strings. The source-strings specify new program fragments for which the corresponding AST fragment can always be obtained and inserted into an existing AST. To allow semantics based transformations, which require the full type information of a given program, we make the type information of the program available as annotations of the AST. The availability of the full type information is crucial to allow semantics based transformations as we shall demonstrate in the following example.

The abstract grammar describes the set of all ASTs. Because we do not use multiple inheritance, the class hierarchy of the object-oriented AST forms a tree. The abstract grammar is designed such that it directly corresponds to the class hierarchy and the successor information of AST nodes. Inner nodes of the class hierarchy tree correspond to non-terminals in the grammar whereas outer nodes (leafs) correspond to terminals in the grammar. The correspondence is made explicit by using the class names as names for terminals and non-terminals respectively.

Our present version of the default abstract grammar for full C++ has 165 rules. All annotated AST information gathered by the frontend at each AST node is available through a variable `astNode`. The variable always holds a pointer to the corresponding AST node of a parsed terminal. Information available is type information for every expression and declaration, line and column information of the original program, etc.

In the following example we show how the attribute grammar in combination with the use of source-strings and AST replacement operations, allows to specify the introduction of OpenMP pragmas and the transformation of for-loops to conform to the required canonical form of an omp parallel for.

In the example source in fig. 6 we show an iteration on a user-defined container with an iterator. This pattern is frequently used in applications using C++98 standard container classes. The object `f` is an instance of the user-defined class `Foo`. The transformation we present takes into account the semantics of the

Before transformation

```
Foo f; list l;
...
for (list::iterator i = l.begin(); i != l.end(); i++) {
  f.foo(*i);
}
```

After transformation

```
Foo f; list l;
...

// Build the supporting container
SupportingOmpContainer_list l2 (l);

#pragma omp parallel for
for (int i = 0; i < l2.size(); i++) {
  f.foobar( l2[i] );
}
```

Fig. 6. An iteration on a user-defined container `l` that provides an iterator interface. The object `f` is an instance of the user-defined class `Foo`. Object `l` is of type `list`. In the optimization the iterator is replaced by code conforming to the required canonical form of an OpenMP parallel for.

type `Foo` and the semantics of class `list`. The transformation is therefore specific to these classes and its semantics.

For the type `list` we know that the type `iterator` defined in the class follows the iterator pattern as used in the STL. For the type `Foo` we know that the method `f` is thread safe. We show the core of a transformation to transform the code into the canonical form of a for-loop as required by the OpenMP standard. We also introduce the OpenMP pragma directive. Note that the variable `i` in the transformed code is implicitly private according to the OpenMP standard 2.0 .

The test, `isUserDefIteratorForStatement`, to determine whether the transformation can be applied, is conservative. It might not always allow to perform the optimization although it would be correct but it is never applied when we cannot ensure that the transformed code would be correct.

In the example in fig. 7 the grammar rule of `SgScopeStatement` is shown. The terminal `SgForStatement` in the example corresponds to the class `SgForStatement`. The semantic actions associated with this rule are executed whenever a node of type SgForStatement is parsed. The variable `astNode` is a pointer to the respective AST node of the terminal and assigned by our supporting system when the scanner accesses the token stream. Note that every terminal in the grammar corresponds to a node in the AST, except the parentheses.

Methods of the object `subst` allow to insert new source code and delete subtrees in the AST. The substitution object `subst` buffers pairs of target location and string. The substitution is not performed before the semantic actions of all subtrees of the target location node have been performed. This mechanism allows to check whether substitutions would operate on overlapping subtrees of

the AST (in the same attribute evaluation). In case of overlapping subtrees an error is reported.

The object `query` is of type `AstQuery` and provides frequently used methods for obtaining information stored in annotations of the AST. These methods are also implemented as attribute evaluations.

```
SgScopeStatement<bool isOmpFor>
  = SgForStatement
      (.
        isOmpFor
        = ompTransUtil.isUserDefIteratorForStatement(astNode,isOmpFor);
      .)
    "(" SgForInitStatementNT<isOmpFor> SgExpressionRootNT
        SgExpressionRootNT SgBasicBlockNT<isOmpFor>
    ")"
      (.
        if(isOmpFor) {
          string iVarName = query.iteratorVariableName(astNode);
          string iContName = query.iteratorContainerName(astNode);
          string iContType = query.iteratorContainerType(astNode);
          string parTypeName = ompTransUtil.supportingParType(astNode,iContType);
          string parContName = ompTransUtil.uniqueVarName(astNode,iContName);
          string modifiedBodyString
                 = ompTransUtil.derefToIndexBody(astNode,iVarName,iContName);
          string support = parTypeName+" "+parContName+"("+iContName+");\n";
          string beforeForStmt
                 = "#pragma omp parallel for\n";
          string newForStmt = "for( int "+iVarName+"=0;"
                              + iVarName+"<"+parContName+".size();"
                              + iVarName+"++ ) "+modifiedBodyString;
          subst.replace(astNode,support + beforeForStmt + newForStmt);
        }
      .)
  | ...
```

Fig. 7. A part of the grammar rule of SgScopeStatement of the abstract C++ grammar with the semantic action specifying the transformation of a SgForStatement.

The inherited attribute `isOmpFor` is used to handle the nesting of for-loops. It depends on how an OpenMP compiler supports nested parallelism whether we want to parallelize inner for statements or only the outer for statement. In future this decision will be made more specific to OpenMP compilers on different platforms and the boolean attribute will be replaced by an object to provide more information about the context of OpenMP for-loops.

The object `query` of type `AstQuery` offers methods to provide information on subtrees that have been proven to be useful in different transformations. In the example we use it to obtain variable names and type names. The example shows how we can decompose different aspects of a transformation into separate attribute evaluations. The methods of the query object are implemented by using the attribute evaluation.

After a preprocessing step of the grammar file, we use a successor of Coco/R [8], the C/C++ version ported by Frankie Arzu to generate the parser code. Coco/R is a compiler generator which allows to specify a scanner and a parser

in EBNF for context free languages. The grammar has to be LL(1). We use this tool to operate on the token stream of AST nodes. Therefore we do not use the scanner generator capabilities of Coco/R and implemented a scanner to operate on the token stream of AST nodes. The stream is defined by a fixed traversal on the AST. The integration of this parsing tool allowed us to leverage the attribute evaluation capabilities of parsing tools.

In fig. 6 the generated code is shown. The access uses the notation for random access iterators. The `SupportingOmpContainer_list` class is used to generate an array of pointers to all elements of the list to achieve a complexity of O(1) for the element access. The list of pointers is generated when the supporting container `l2` is created. When the generated code is compiled with an OpenMP compiler the body is executed in parallel.

Note that the generated source code can have a slightly different formatting because the format of the source code is a beautified version of the source corresponding to the transformed AST.

4 Translation of OpenMP Directives

To generate code that explicitly manages parallelism, we use ROSE to build a specialized source-to-source translator that transforms OpenMP `parallel for` directives into explicit calls to an OpenMP runtime library [9]. For our work, we have selected the Nanos OpenMP runtime library [10]. We are in the process of adding support for additional OpenMP constructs to our infrastructure. Alternatively, we can unparse the original source code with OpenMP directives and use the resulting source code as input to a commercial OpenMP C++ compiler to generate parallel code [11–14].

5 Related Work

The research community has developed many automatic parallelizing compilers. Examples of these research compilers include the DSystem [15], the Fx compiler [16], the Vienna Fortran Compiler [17], the Paradigm compiler [18], the Polaris compiler [19], and the SUIF compiler [20]. However, except for SUIF, which has frontends for Fortran, C, and C++; the others listed above optimize only Fortran applications. By providing a C++ frontend for automatic parallelization, we complement previous research in providing support for higher-level object-oriented languages. In addition, we extend previous techniques by utilizing the semantic information of user-defined containers and thus allowing user-defined abstractions to be treated as part of a domain-specific language.

As more and more programmers are using OpenMP to express parallelism, many OpenMP supporting compilers were developed, including both research projects [10, 21–23] and commercial compilers [11–14]. In addition to OpenMP-directive translation, many research compiler infrastructures also investigate techniques to automatically generate OpenMP directives and to optimize the parallel execution of OpenMP applications. However, these research compilers

only support applications written in C or FORTRAN, while existing commercial C++ compilers target only specific machine architectures and do not provide an open source-to-source transformation interface to the outside world. By providing a flexible source-to-source translator, we complement previous research by presenting an open research infrastructure for optimizing C++ constructs and OpenMP directives.

6 Conclusions and Future Work

This paper presents a C++ infrastructure for semantic-driven parallelization of computations that operate on user-defined containers that have an access interface similar to that provided by the Standard Template Library in C++. First, we provide an interface for library developers to inform our compiler about the semantics of their containers and the side-effects of their library functions. Then, we use this information to parallelize loops that iterate over these containers automatically when it is safe to do so.

Our analysis algorithm conservatively disallows the parallelization of loops that modify non-local memory locations, that is, memory locations that are not elements of the user-defined container and are defined outside of the loop. In the future, we will extend our algorithm to be more precise by incorporating global alias analysis and array dependence analysis techniques [7]. This more sophisticated algorithm will be as precise as those used by other automatic parallelizing compilers [15–20], while still being more aggressive for user-defined abstractions by optimizing them as part of a domain-specific language.

References

1. Cooper K. Dongarra J. Fowler R. Gannon D. Johnsson L. Kennedy K. Mellor-Crummey J. Torczon L. Broom, B. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 2000.
2. Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.
3. Daniel Quinlan, Brian Miller, Bobby Philip, and Markus Schordan. Treating a user-defined parallel library as a domain-specific language. In *16th International Parallel and Distributed Processing Symposium (IPDPS, IPPS, SPDP)*, pages 105–114. IEEE, April 2002.
4. Daniel Quinlan, Markus Schordan, Brian Miller, and Markus Kowarschik. Parallel object-oriented framework optimization. *Special Issue of Concurrency: Practice and Experience*, 2003, to appear.
5. Edison Design Group. http://www.edg.com.
6. Francois Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An object-oriented toolkit and class library for building fortran and C++ restructuring tools. In *Proceedings. OONSKI '94*, Oregon, 1994.
7. R. Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, October 2001.

8. Hanspeter Moessenboeck. Coco/R - A generator for production quality compilers. In *LNCS477, Springer*, 1991.
9. Daniel Quinlan, Markus Schordan, Qing Yi, and Bronis de Supinski. A C++ infrastructure for automatic introduction and translation of OpenMP directives. In *WOMPAT'03: OpenMP Shared Memory Parallel Programming, International Workshop on OpenMP Applications and Tools*, volume 2716 of *Lecture Notes in Computer Science*, pages 13–25. Springer Verlag, June 2003.
10. Eduard Ayguade, Marc Gonzalez, and Jesus Labarta. Nanoscompiler: A research platform for openMP extensions. In *European Workshop on OpenMP*, September 1999.
11. Silican Graphics Inc. *Optimizing Compilers for High-Performance Computing.* www.sgi.com/developers/devtools/languages/mipspro.html.
12. IBM. *VisualAge C++ Professional for AIX V6.0.* www-1.ibm.com/servers/eserver/ecatalog/us/software/6146.html.
13. Xinmin Tian, Aart Bik, Milind Girkar, Paul Grey, Hideki Saito, and Ernesto Su. Intel openMP C++/Fortran compiler for hyper-threading technology: Implementation and performance. *Intel Technology Journal*, 6(1):36–46, 2002.
14. Fujitsu. *Fortran & C Packages for SPARC Solaris.* www.fr.fse.fujitsu.com/devuk/solaris.shtml.
15. V. Adve, G. Jin, J. Mellor-Crummey, and Q. Yi. High performance fortran compilation techniques for parallelizing scientific codes. In *Proceedings of SC98: High Performance Computing and Networking*, Nov 1998.
16. J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *Proc. of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, San Diego, May 1993.
17. S. Benkner. Vfc: The vienna fortran compiler.
18. P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The paradigm compiler for distributed-memory message passing multicomputers. In *in Proceedings of the First International Workshop on Parallel Processing*, Bangalore,India, Dec 1994.
19. D. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin". Polaris: A new-generation parallelizing compiler for mpp's. Technical Report 1306, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. and Dev., june 1993.
20. M. S. Lam S. P. Amarasinghe, J. M. Anderson and C. W. Tseng. The suif compiler for scalable parallel machines. In *in Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, Feb 1995.
21. Christian Brunschen and Mats Brorsson. OdinMP/CCp - a portable implementation of openMP for c. In *European Workshop on OpenMP*, September 1999.
22. Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka. Design of openMP compiler for an SMP cluster. In *European Workshop on OpenMP*, September 1999.
23. Seung Jai Min, Seon Wook Kim, Michael Voss, Sang Ik Lee, and Rudolf Eighmann. Portable compilers for openMP. In *Workshop on OpenMP Applications and Tools*, July 2001.