A C++ Infrastructure for Automatic Introduction and Translation of OpenMP Directives

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

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

Abstract. In this paper we describe a C++ infrastructure for source-to-source translation. We demonstrate the translation of a serial program with high-level abstractions to a lower-level parallel program in two separate phases. In the first phase OpenMP directives are introduced, driven by the semantics of high-level abstractions. Then the OpenMP directives are translated to a C++ program that explicitly creates and manages parallelism according to the specified directives. Both phases are implemented using the same mechanisms in our infrastructure.

1 Introduction

The use of OpenMP within the OpenMP research community seems complicated by the lack of easy to use compiler infrastructure. Although much work is focused on OpenMP for FORTRAN 77 and FORTRAN 90, and there may be an abundance of C language compiler infrastructure; the unavailability of C++ compiler infrastructure has significantly limited the many research opportunities. In this paper, we present a useful infrastructure, ROSE [1], to assist the OpenMP research community generally, but particularly for OpenMP/C++ research.

Our infrastructure allows the automated introduction of OpenMP directives based on the semantics of user-defined abstractions. The introduction of pragmas, when adding OpenMP directives to a given code, is one of many possible applications. Another one is the translation of OpenMP directives; the recognition of specific pragma directives and the translation of associated code fragments to generate a program that explicitly creates and manages parallelism. We shall use a running example to illustrate both phases and how the ROSE infrastructure [1] can simplify these tasks. Through this example, we demonstrate the relatively simple specification of an OpenMP transformation to use the lower level Nanos Library for OpenMP [2]. We also discuss how to modify that transformation to implement the full OpenMP standard. Given the semantic similarity

** This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

between most OpenMP runtime libraries, we expect that transformations for other OpenMP runtime libraries should be equally simple.

Since within ROSE we have the full type resolution within the AST, and not just syntax, the type information of specific user-defined types can be used as a basis for the optimization of applications that use them. And by including knowledge of the semantics of specific abstractions, fundamentally more information is available to the compiler and greater levels of optimization are often possible, depending upon the abstractions. We will show through the use of an array abstraction, that because the stronger array semantics is satisfied by the weaker OpenMP constraints we can automate the introduction of OpenMP directives into otherwise serial code. This approach permits fundamentally serial code to use the additional semantics of the array abstractions and be run as parallel code.

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 intermediate representation. Several different components can be used to build the midend of a translator: to operate on the AST, a predefined traversal mechanism, a restructuring mechanism, and an attribute evaluation mechanism can be used to implement a transformation. Other features are for example 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 fig. 1).

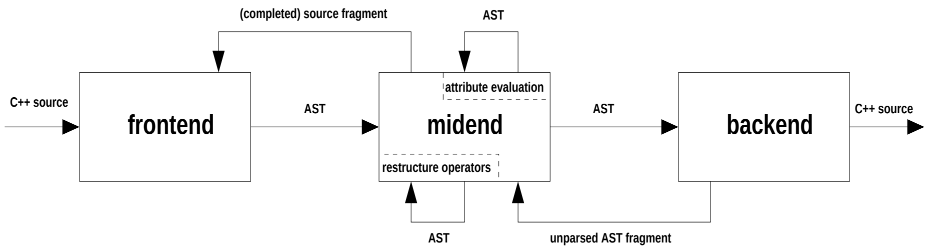

Fig. 1. ROSE Source-To-Source infrastructure with frontend/backend reinvocation

2.1 Frontend

We use the Edison Design Group C++ frontend (EDG) [3] to parse C++ programs. The EDG frontend generates an AST and performs a full type evaluation of the C++ program. The AST is represented as a C data structure. We translate this data structure into an object-oriented abstract syntax tree which is used by the midend as intermediate representation.

2.2 Midend

The midend supports restructuring of the AST. Code that is added to the AST can be specified as a source string, using C++ syntax, or by constructing subtrees node by node. An AST restructuring operation specifies a location in the AST where code should be inserted, deleted, or replaced. The code can be specified as C++ source string or an AST subtree. A program transformation consists of a series of AST restructuring operations.

The order of the restructuring operations is based on a pre-defined traversal. In a transformation the AST is traversed and different restructuring operations are invoked on the AST. The problem of restrucuring the AST while traversing it, is addressed by making 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 before the traversal of this subtree is finished. We provide interfaces for invoking restructuring operations that buffer these operations to ensure that no subtrees are replaced while they are traversed.

The attribute evaluation mechanism allows the computation of attribute values for AST nodes. Context information can be passed down the AST as inherited attributes and results of computations on a subtree can be computed as synthesized attributes (passing information upwards the tree). Examples for values of inherited and synthesized attributes are type information, size of arrays, the nesting level of loops, the scopes of associated pragma statements, etc. These values can be used to specify constraints on a transformation, i.e. to decide whether a restructuring operation should be applied.

Our infrastructure allows to use C++ source code strings to define code fragments. Any source string which 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. The string, s, is extended by our system to form a complete program. This completed program is parsed into an AST by reinvoking the frontend. From this AST, we extract the AST fragement that corresponds to the source string s. This AST fragement is inserted into the AST.

2.3 Backend

The AST is unparsed and C++ source code is generated. It can be specified to 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 generated methods.

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 Semantics-Driven Introduction of OpenMP Directives

The use of high-level abstractions so greatly improves the productivity of developing scientific applications that we seek a way to address the numerous performance issues associated with it.

3.1 User-Defined Abstractions

User-defined abstractions permit a way to tailor the user-environment to be more domain specific than a general purpose language could allow. General purpose languages are expensive to develop and result from many years of work. The compilers that define the language are both expensive and difficult to develop. Such an investment is only possible for a sufficiently large user group.

Simplifying the development of many applications within a specific domain is commonly done through the development of domain-specific libraries. The libraries invariably define abstractions that hide numerous tedious details associated with the development of applications within a specific domain. The combination of a general purpose language and a domain specific library is not the same as a domain-specific language. The essential difference is that the complete semantics of a library's abstractions are unknown at compile time and, thus, some significant optimizations are impossible for the compiler to implement. The result is all too often that many essential abstractions are abandoned because they can't provide sufficiently high performance.

3.2 A++/P++ Serial and Parallel Array Class Library

We use a motivating example from the A++/P++ array class library [4] to show how the ROSE framework can be used by the library writer to develop a source-to-source translator that optimizes code based on high-level semantics. The example uses two classes which are implemented twice; once in the serial A++ library and again in the parallel P++ library. Within our motivating example we consider the following trivial five-point stencil array operation. In figure 2, `A` and `B` are multidimensional array objects of type `floatArray`. `I` and `J` are `Range` objects that together specify a two dimensional index space of the arrays `A` and `B`. The following sections demonstrate how ROSE supports the optimization of a scientific application code through our running example.

3.3 Automated Insertion of OpenMP Directives

Because of the parallel semantics of the A++ and P++ array objects, their use is interchangeable. This permits serial applications to be developed using A++ (serial arrays) and then recompiled to run in distributed memory mode using P++ (parallel arrays). Some simple constraints are that any use of non A++ array objects not constrain the data-parallel model that is hidden within the array semantics.

Since the parallel array semantics of A++ and P++ are consistent with those of OpenMP, OpenMP directives can safely introduce shared memory parallelism into all uses of A++ and P++ array objects. This is essential for the automated insertion of OpenMP directives without complex dependence analysis of the serial code.

3.4 Example C++ Code

The example codes in figure 2 and figure 3 demonstrate the transformation of high-level A++ code to highly efficient OpenMP code. The two codes are semantically equivalent, but the first code shows the use of high-level array abstractions. The semantics of the array abstractions are similar to those of array statements in FORTRAN 90, but the implementation is a (C++) class library instead of a (FORTRAN77) language extension. Clearly, the standard compilation process cannot take the semantics of the array class objects into account since those semantics are user defined. At this high level of abstraction, the C++ compiler is quite powerless to introduce any significant optimizations, precisely because the abstraction's semantics that are relevant to critical optimizations are user-defined and unknown.

```
int n;
Range I,J,K;
floatArray A(n,n,n);
floatArray rhs(n,n,n);
floatArray B(n,n,n);
...
A(I,J,K) = rhs(I,J,K) + ( B(I+1,J,K) + B(I-1,J,K) + B(I,J-1,K) +
           B(I,J+1,K) + B(I,J,K-1) + B(I,J,K+1) - 6.0 * B(I,J,K) );
```

Fig. 2. Example: Code fragment showing the use of A++/P++ array semantics.

The high-level A++ code can be automatically transformed into the greatly expanded, but more efficient code shown in figure 3. The ROSE infrastructure allows the library implementer to leverage the semantics of the array class objects that are required to implement the transformation in a source-to-source translator that provides a library-specific compilation process. Specifically, the ROSE frontend creates an AST. The traversal mechanism allows the targeted array class statements to be located in the code. The restructuring mechanism is used to replace the high-level code with the corresponding, but more efficient code and the attribute mechanism supports important details of the transformation such as proper declaration of the loop control variables. A very small and almost trivial part of the transformation is the additional step to have the transformation also generate the OpenMP directive before the outermost loop.

```
#define SC(x1,x2,x3) /* case UniformSizeUnitStride */ (x1)+(x2)*_size1+(x3)*_size2
#pragma omp parallel for private (_3, _2, _1) \
      shared (AIJKpointer, rhsIJKpointer, BIJKpointer)
for (_3 = 0; _3 < _length3; _3++) {
     for (_2 = 0; _2 < _length2; _2++) {
        for (_1 = 0; _1 < _length1; _1++) {
           AIJKpointer[SC(_1,_2,_3)] =
               rhsIJKpointer[SC(_1,_2,_3)] +
               (BIJKpointer[SC((_1 + 1),_2,_3)] + BIJKpointer[SC((_1 - 1),_2,_3)] +
                BIJKpointer[SC(_1,(_2 - 1),_3)] + BIJKpointer[SC(_1,(_2 + 1),_3)] +
                BIJKpointer[SC(_1,_2,(_3 - 1))] + BIJKpointer[SC(_1,_2,(_3 + 1))] -
                6.0 * BIJKpointer[SC(_1,_2,_3)] );
        }
     }
  }
```

Fig. 3. Example: Transformed A++/P++ array class code fragment showing the insertion of an OpenMP directive (excluding preceding declarations)

3.5 Discussion

The ROSE mechanisms provide a general approach for the optimization of complex libraries that is not specific to the A++/P++ library. We use this example because it is both a high-level abstraction specifically tailored to parallel scientific computing and because it is one with which we are familiar. Improving the performance of the A++/P++ library also has a direct impact on other applications and libraries using it (the Overture Framework [5] in particular).

4 Translation of OpenMP Directives

We use ROSE to build a specialized source-to-source translator that transforms OpenMP directives into lower-level code using an OpenMP runtime library. For our work, we have selected the Nanos OpenMP runtime library [2], but our intention is to demonstrate that any runtime library could be used. We believe our approach would be nearly the same for any OpenMP runtime library, given the seemingly strong semantic resemblance between the few that we have seen. An aspect of our effort is to show how easily other researchers within the OpenMP community could use the ROSE compiler infrastructure for OpenMP research. We hope that access to open compiler infrastructure for C, and particularly for C++, will be found useful.

4.1 Translation Specification

Before translating OpenMP directives into runtime library calls, we must first define a specification that maps the input and output of the translation. Figure 4 presents an example of such mapping, which translates the OpenMP `parallel-for` directive (with the `shared`, `private`, `default` and `schedule` clauses) into calls to the lower-level Nanos OpenMP runtime library [2]. We choose the `parallel-for` directive because it is suitable for illustrating our OpenMP source-to-source translator (shown in Figure 5) and because the ROSE

```
Input:
   #pragma omp parallel for schedule($scheduletype, $chunksize) default ($defaulttype) \
                              shared($shared_var_list) private($private_var_list)
   for ($i = $lb; $i <= $ub; $i + = $step) {
      $loop_body
   }

Output:
   void supportingOpenMPFunction$id( int* intone_me_01, int* intone_nprocs_01,
                                     int* intone_master01, $shared_var_decl_list )
   {
      $private_var_decl_list;
      int intone_start, intone_end, intone_last;

      intone_begin_for($lb, $ub, $step, $chunksize, $scheduletype);
      while (intone_next_iters( &intone_start, &intone_end, &intone_last)) {
            for ($i = intone_start; $i <= intone_end-1; $i + = $step) {
                  $loop_body
            }
      }
      intone_end_for(true)
   }

   int intone_nprocs_01 = intone_cpus_current();
   intone_spawnparallel( supportingOpenMPFunction$id, $numOfArgs, intone_nprocs_01,
                         $shared_var_list);
```

Fig. 4. Specification for translating the OpenMP parallel-for directive into Nanos runtime library calls (the bold text marks OpenMP keywords, and the $ sign denotes parameters of the input and output fragments.)

infrastructure can automatically introduce it using the A++/P++ array semantics, as shown in Figure 3. After applying the mapping in Figure 4, our OpenMP source-to-source translator can further transform the OpenMP code in Figure 3 into the Nanos runtime library calls; the result is shown in Figure 6.

In general, to provide translation support for the entire set of OpenMP directives, we need to specify a translation mapping, such as the one in Figure 4, for each OpenMP directive. These mappings should be easily constructed from the manual of an OpenMP runtime library. We then use these mappings to instantiate the general translation algorithm in Figure 5. Though currently we have implemented only the translation of the `parallel-for` directive within the ROSE infrastructure, other OpenMP directives can be translated in a similar fashion.

4.2 Translation Algorithm

Figure 5 presents the structure of a ROSE source-to-source translator that transforms an arbitrary OpenMP directive into its corresponding runtime library calls. This source-to-source translator is separated into the following three phases.

The first phase uses the front end of ROSE to parse the input program into an AST, which provides support for most C++ high-level constructs and thus closely matches the structure of the original program. Within the same phase, the source-to-source translator then makes a second pass of the constructed AST

```
(1)Parse the C++/C input program and construct an Abstract Syntax Tree
   Parse the OpenMP directives in the constructed AST
(2)Traverse the Abstract Syntax Tree of the input program
   At each tree node astNode:
      if ( (pragma = PrevStatement(astNode)) is an OpenMP directive)
         string OpenMP_support_func = parameterized supporting-function string for pragma
         for (each parameter par in OpenMP_support_func)
            string par_val = Compute-Parameter-Value(par,astNode)
            String-Replace-Substring(OpenMP_support_func, par, par_val)
         Add OpenMP_support_func into global scope
         OpenMP_replace_pragma = parameterized intone_spawnparallel call for pragma
         Substitute parameters in OpenMP_replace_pragma with correct values
         replace pragma and astNode subtrees with OpenMP_replace_pragma
(3)Unparse the Abstract Syntax Tree
```

Fig. 5. Algorithm for translating OpenMP directives into runtime library within the ROSE infrastructure

to expand the OpenMP directives. Unlike the C++ front end, the OpenMP construct parser is not already implemented in ROSE and thus needs to be provided by the OpenMP source-to-source translator. It is our plan to provide a full implementation of this parser within our OpenMP source-to-source translator.

The OpenMP construct parser not only translates each string pragma into structured AST nodes, it also automatically collects all the implicit parallelization information pertinent to the OpenMP directive. For example, after this pass, even if the `parallel-for` directive in Figure 4 does not have a `shared` clause (assuming all variables are shared by default), the OpenMP parser will automatically collect the set of shared variables and then insert a `shared` clause into the parsed pragma. The exact behavior for variables in either $shared_var_list or $private_var_list is determined by the `default` clause (if present) and is implemented entirely in the OpenMP parser. Thus, the subsequent phases of the translation algorithm can assume that all data storage attributes are explicit (this is equivalent to having a `default (none)` clause in the original worksharing construct).

The second phase of the OpenMP source-to-source translator then traverses the AST and transforms the fully expanded OpenMP directives within the AST. At each node *astNode*, if the statement *pragma* immediately before *astNode* is an OpenMP directive, we translate this directive by first constructing a supporting function (*OpenMP_support_func*) for the original code (the subtree rooted at *astNode*). This supporting function is a parameterized string provided by the translation mapping specification (e.g., the section output in Figure 4). We then proceed to substitute all the parameters in the supporting-function string with their corresponding string values pertinent to the original code. Since the source-to-source translator has the pre-knowledge about all the parameters in the *OpenMP_support_func* string, it can compute the values for these parameters by invoking pre-defined AST analysis facilities within ROSE. We then insert the fully expanded *OpenMP_support_func* into the global scope and thus make it another function definition of the original C++ program. Next, we create a string, *OpenMP_replace_pragma*, that invokes the expanded supporting func-

```
void supportingOpenMPFunction__0_0( int* intone_me_01, int* intone_nprocs_01,
   int* intone_master_01, float * AIJKpointer, float * rhsIJKpointer,
   float * BIJKpointer, int _length1, int _length2, int _size1, int _size2)
   {
     int _1, _2, _3;
     int intone_start, intone_end, intone_last;
     intone_begin_for(0,100,1,0,0);
     while(intone_next_iters(&intone_start,&intone_end,&intone_last)) {
          for (_3 = intone_start; _3 <= intone_end; _3++) {
               for (_2 = 0; _2 < _length2; _2++) {
                   for (_1 = 0; _1 < _length1; _1++) {
                        AIJKpointer[_1 + _2 * _size1 + _3 * _size2] =
                        rhsIJKpointer[_1 + _2 * _size1 + _3 * _size2] +
                         (BIJKpointer[(_1 + 1) + _2 * _size1 + _3 * _size2] +
                          BIJKpointer[(_1 - 1) + _2 * _size1 + _3 * _size2] +
                          BIJKpointer[_1 + (_2 - 1) * _size1 + _3 * _size2] +
                          BIJKpointer[_1 + (_2 + 1) * _size1 + _3 * _size2] +
                          BIJKpointer[_1 + _2 * _size1 + (_3 - 1) * _size2] +
                          BIJKpointer[_1 + _2 * _size1 + (_3 + 1) * _size2] -
                          6.0 * BIJKpointer[_1 + _2 * _size1 + _3 * _size2] );
                      }
                 }
             }
        }
     intone_end_for(true);
   }
intone_nprocs_01 = intone_cpus_current();
intone_spawnparallel( supportingOpenMPFunction__0_0, 8, intone_nprocs_01, AIJKpointer,
                      rhsIJKpointer, BIJKpointer, _length1,_length2,_size1,_size2);
```

Fig. 6. Example: transformed A++/P++ array class code fragment using the Nanos runtime library

tion using parallel threads (e.g., the *intone_spawnparallel* call in Figure 4). Finally, after substituting the parameters in *OpenMP_replace_pragma* with corresponding values, we use *OpenMP_replace_pragma* to replace both the original OpenMP directive (*pragma*) and the original code fragment (the subtree rooted at *astNode*).

Most steps described above can be realized in a straightforward fashion by simply invoking existing ROSE mechanisms. To illustrate the simplicity of this mapping, Figure 7 presents the ROSE C++ implementation for translating the `parallel-for` directive defined in Figure 4. Here we omit some parameter substitutions due to lack of space. Note that ROSE provides facilities to directly edit parameters in strings and to insert strings directly into the AST (they are parsed into *abstract syntax subtrees* before being inserted into the global AST).

As the final phase, after all the OpenMP directives have been translated, the source-to-source translator unparses the transformed AST to produce a C++ program that includes only calls to the OpenMP runtime library.

4.3 Discussion

Generalizing the source-to-source translator discussed in the preceding sections to provide support for the full OpenMP specification is the subject of on-going work. In this section, we discuss the modifications that our approach requires to

```
OpenMPSynthesizedAttribute
OpenMPTraversal::evaluateRewriteSynthesizedAttribute (
                       SgNode* astNode, OpenMPInheritedAttribute inheritedAttribute,
                       SubTreeSynthesizedAttributes synthesizedAttributeList ) {
  OpenMPSynthesizedAttribute returnAttribute(astNode);
  if ( OmpUtility::isOmpParallelFor(astNode) ) {
    SgForStatement *forStatement = isSgForStatement(astNode);
    string supportFunction = " \n\
void supportingOpenMPFunction_$ID ( int* intone_me_01, int* intone_nprocs_01,
                                    int* intone_master01, $SHARED_VAR_DECL_LIST ) { \n\
    $PRIVATE_VAR_DECL_LIST; \n\
    int intone_start, intone_end, intone_last; \n\
    intone_begin_for($LB,$UB,$STEP,$CHUNKSIZE,$SCHEDULETYPE); \n\
    while ( intone_next_iters(&intone_start,&intone_end,&intone_last)) { \n\
      for ($LOOPINDEX = intone_start; $LOOPINDEX <= intone_end; $LOOPINDEX += $STEP) { \n\
        $LOOP_BODY; \n\
      } \n\
    } \n\
    intone_end_for(true); \n\
  } \n";
    string spawnParallel = " \
intone_nprocs_01 = intone_cpus_current(); \n\
intone_spawnparallel(supportingOpenMPFunction_$ID,$NUM_ARGS,intone_nprocs_01,\
$SHARED_VAR_LIST);\n";

    // Edit the function name and define a unique number as an identifier
    string uniqueID = buildUniqueFunctionID();
    supportFunction = StringUtility::copyEdit(supportFunction, "$ID",uniqueID);
    spawnParallel = StringUtility::copyEdit( spawnParallel, "$ID",uniqueID);

    // Edit the loop parameters into place
    string loopBody = forStatement->get_loop_body()->unparseToString();
    supportFunction = StringUtility::copyEdit(supportFunction, "$LOOP_BODY",loopBody);
    ... // similar copyEdits for $LOOPINDEX, $LB, $UB, $STEP

    // Edit the OpenMP parameters into place
    OmpUtility ompData (astNode);
    string privateVarDeclList = ompData.generatePrivateVariableDeclaration();
    string sharedVarList = ompData.generateSharedVariableFunctionParameters();
    string sharedVarDeclList = ompData.generateSharedVariableFunctionDeclarations();
    supportFunction = StringUtility::copyEdit(supportFunction,
                       "$SHARED_VAR_DECL_LIST",sharedVarDeclList);
    supportFunction = StringUtility::copyEdit(supportFunction, "$SHARED_VAR_LIST",
                       sharedVarList);
    spawnParallel = StringUtility::copyEdit(spawnParallel,
                       "$SHARED_VAR_LIST",sharedVarList);
    supportFunction = StringUtility::copyEdit(supportFunction,
                       "$PRIVATE_VAR_DECL_LIST",privateVarDeclList);
    ... //  similar copyEdits for $CHUNKSIZE,$SCHEDULETYPE, and $NUM_ARGS

    AST_Rewrite::addSourceCodeString(returnAttribute, "#include \"nanos.h\"",
                       inheritedAttribute, AST_Rewrite::GlobalScope,
                       AST_Rewrite::TopOfScope, AST_Rewrite::TransformationString, false);
    AST_Rewrite::addSourceCodeString( returnAttribute, supportFunction, inheritedAttribute,
                       AST_Rewrite::GlobalScope, AST_Rewrite::BeforeCurrentPosition,
                       AST_Rewrite::TransformationString, false);
    AST_Rewrite::addSourceCodeString( returnAttribute, transformationVariables,
                       inheritedAttribute, AST_Rewrite::LocalScope, AST_Rewrite::TopOfScope,
                       AST_Rewrite::TransformationString, false);
    AST_Rewrite::addSourceCodeString( returnAttribute, spawnParallel, inheritedAttribute,
                       AST_Rewrite::LocalScope, AST_Rewrite::ReplaceCurrentPosition,
                       AST_Rewrite::TransformationString, false);
  }
  return returnAttribute;
}
```

Fig. 7. Example: Code fragment showing translation of an OpenMP directive.

provide that support. We consider all OpenMP directives, including any associated clauses.

The source-to-source translator presented thus far implements the OpenMP `parallel-for` construct, including the `private`, `shared`, `default` and `schedule` clauses. The source-to-source translator, as described, does not implement several possible clauses of the directive; extending it to support the remaining clauses is straightforward. As discussed in section 4.2, parsing of the construct determines the lists of private and shared variables, including those for which the storage attribute is implicit. The construct parsing can easily be modified to build lists for the other data attribute clauses. As discussed in the Nanos documentation [6], variables with the `firstprivate` and `lastprivate` attributes become arguments to the call of the supporting function with corresponding internal variable names for the parameters. The only other change necessary to our source-to-source translator is to include the appropriate assignment between the internal variable name and the name used in the loop body in the supporting function string. The `reduction` clause requires similar changes, with the assignment guarded by a lock that is initialized prior to spawning the parallel region. The `if` clause requires that $OpenMP\_replace\_pragma$ be extended to include the *intone_spawnparallel* call in an if statement with the original code cloned into the *else* clause, which is easily implemented with the ROSE restructuring mechanism.

Changes to the source-to-source translator that would support splitting the combined `parallel-for` directive are not difficult. In order to support the OpenMP `parallel` construct (i.e., without the *for* loop), the string used for the supporting function would only include the portions that establish the variable lists and the original code. We can support stand-alone OpenMP `for` constructs by replacing the pragma and original code with the body of the supporting function instead of the *intone_spawnparallel* call. In order to implement orphaned directives correctly with separate compilation, the runtime library must support this in-place replacement.

Straightforward modifications to the source-to-source translator will also extend it to implement the other work-sharing constructs and synchronization directives. The Nanos documentation discusses how to implement the `sections` construct and the `single` directive as variations of the `for` construct, while the replacement code for the synchronization constructs are even simpler. Although we could modify the replacement code to use other calls for runtime libraries that provide calls specific to the `sections` construct and the `single` directive, we plan to implement them as variants of the `for` construct initially.

We have not fully determined how to support `threadprivate` storage in our source-to-source translator. Our support for `threadprivate` storage is highly dependent on the support provided by the OpenMP run time library. The Nanos runtime library targets FORTRAN, and uses pseudo-dynamically allocated storage. More straightforward solutions are possible in C and C++ and one option is to provide an alternative mechanism. Whether or not we use the existing support of the runtime library, we expect that providing support for `threadprivate` storage will be fairly straightforward if it has static block-scope; while the support may be more complex for file-scope or name-space scope, particularly for initializating the storage.

The generality of the OpenMP translation in Figure 5 and the just discussed modifications depends on specific design properties of the OpenMP runtime library. In particular, given an OpenMP runtime library implementation, if a translation interface similar to Figure 4 can be defined for each OpenMP directive, the source-to-source translator can easily be adapted to provide all the necessary translation support. Otherwise, if the translation of a particular OpenMP directive not only depends on itself and the source code that it applies to, but also depends on the subtle variations of its enclosing context, the algorithm in Figure 5 may not be directly applicable.

An example is the treatment of OpenMP `threadprivate` clauses. If the translation interface requires the OpenMP source-to-source translator to generate different output code patterns depending on whether or not `threadprivate` storage has been previously used, a straightforward adaptation of Figure 5 will not work. For such cases, more complicated global analysis and transformation techniques are required.

5 Related Work

Although a number of compilers were developed to support OpenMP applications, most OpenMP research projects [2, 7–9] only support applications written in C or FORTRAN. Because commercial C++ compilers, such as the SGI MIPSpro [10], the IBM XL [11], the Intel KAI Guide [12], and the Fujitsu for SPARC Solaris [13], target specific machine architectures and do not provide an open source-to-source transformation interface to the outside world, they cannot be used by the research community directly to plug in different OpenMP implementations. As the result, no OpenMP source-to-source translator was available for research into optimizing C++ applications. By providing a flexible source-to-source translator, we present an open research infrastructure for optimizing C++ constructs and OpenMP directives.

Previous research source-to-source translators provide various infrastructures for optimizing OpenMP directives. In particular, the OdinMP/CCp compiler [7] takes a C-program with OpenMP directives and produces a C-program for POSIX threads. In contrast, the Omni compiler [8] translates the OpenMP pragmas in C-programs into runtime library calls, which in turn then invoke either POSIX or Solaris threads. The NanosCompiler [2] and the Polaris compiler [9] translate Fortran programs with OpenMP directives in a similar fashion as the Omni compiler. In addition to OpenMP-directive translation, most of these infrastructures also investigate techniques to automatically generate OpenMP directives and to optimize the parallel execution of OpenMP applications. We complement the previous research by presenting an infrastructure for the C++ OpenMP pragma translation and for the automatic generation and optimization of C++ parallel applications.

6 Conclusions and Future Work

We have presented infrastructure for the transformation of C and C++ applications. We have used the semantics of high-level abstractions to demonstrate the automated introduction of OpenMP directives to parallelize serial codes. Finally we demonstrated the translation of a representative OpenMP directive using the Nanos library.

In future work we will make available the OpenMP translation phase as a separate component. This will permit anyone defining transformations to specify them more simply via OpenMP directives and to then process the AST to generate the final code automatically using an OpenMP runtime library.

We are considering applying the ROSE infrastructure to the optimization of the use of OpenMP runtime libraries. This third aspect of ROSE-based OpenMP support would be similar to the A++/P++ source-to-source translator in that it would optimize library use, based domain-specific semantics. For example, we could specialize the use of the Nanos runtime library for special cases for which commercial compilers yield significant performance gains, such as when the number of threads is set to one.

References

1. 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.
2. Eduard Ayguade, Marc Gonzalez, and Jesus Labarta. Nanoscompiler: A research platform for openMP extensions. In *European Workshop on OpenMP*, September 1999.
3. Edison Design Group. http://www.edg.com.
4. R. Parsons and D. Quinlan. A++/P++ array classes for architecture independent finite difference computations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, April 1994.
5. Federico Bassetti, David Brown, Kei Davis, William Henshaw, and Dan Quinlan. OVERTURE: An object-oriented framework for high-performance scientific computing. In *Proceedings of Supercomputing'98 (CD-ROM)*, Orlando, FL, November 1998. ACM SIGARCH and IEEE. Los Alamos National Laboratory.
6. Centre Europeu de Parallelism de Barcelona, Spain. *Nanos Manual.* http://nereida.deioc.ull.es/html/nanos.html.
7. Christian Brunschen and Mats Brorsson. OdinMP/CCp - a portable implementation of openMP for c. In *European Workshop on OpenMP*, September 1999.
8. Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka. Design of openMP compiler for an SMP cluster. In *European Workshop on OpenMP*, September 1999.
9. 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.
10. Silican Graphics Inc. *Optimizing Compilers for High-Performance Computing.* www.sgi.com/developers/devtools/languages/mipspro.html.

11. IBM. *VisualAge C++ Professional for AIX V6.0.* www-1.ibm.com/servers/eserver/ecatalog/us/software/6146.html.
12. 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.
13. Fujitsu. *Fortran & C Packages for SPARC Solaris.* www.fr.fse.fujitsu.com/devuk/solaris.shtml.