ompparser: A Standalone and Unified OpenMP Parser

Anjia Wang[1], Yaying Shi[1], Xinyao Yi[1], Yonghong Yan[1], Chunhua Liao[2], and Bronis R. de Supinski[2]

[1] University of South Carolina, Columbia, SC 29208, USA
{anjia,yaying,xinyaoy}@email.sc.edu, yanyh@cse.sc.edu
[2] Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
{liao6,bronis}@llnl.gov

Abstract. OpenMP has been quickly evolving to meet the insatiable demand for productive parallel programming on high performance computing systems. Creating a robust and optimizing OpenMP compiler has become increasingly challenging due to the expanding capabilities and complexity of OpenMP, especially for its latest 5.0 release. Although OpenMP's syntax and semantics are very similar between C/C++ and Fortran, the corresponding compiler support, such as parsing and lowering are often separately implemented, which is a significant obstacle to support the fast changing OpenMP specification. In this paper, we present the design and implementation of a standalone and unified OpenMP parser, named ompparser, for both C/C++ and Fortran. ompparser is designed to be useful both as an independent tool and an integral component of an OpenMP compiler. It can be used for syntax and semantics checking of OpenMP constructs, validating and verifying the usage of existing constructs, and helping to prototype new constructs. The formal grammar included in ompparser also helps interpretation of the OpenMP standard. The ompparser implementation supports the latest OpenMP 5.0, including complex directives such as metadirective. It is released as open-source from `https://github.com/passlab/ompparser` with a BSD-license. We also demonstrate how it is integrated with the ROSE's open-source OpenMP compiler.

Keywords: OpenMP · Parser · Intermediate Representation · Compiler

1 Introduction

To meet the demand of productive parallel programming on existing and emerging high-performance computing systems, the OpenMP standard has been evolving significantly in recent years [10]. Since the creation of the standard in 1997 that specified a handful of directives, substantial amount of new constructs have been introduced and most existing APIs have been enhanced in each revision. The latest version of OpenMP 5.0, released in 2018, has more than 60 directives. Compiler support thus requires more efforts than before [5]. Compilation of OpenMP programs for both C/C++ and Fortran includes parsing, syntax

and semantics checking, generation of compiler intermediate representation (IR) of OpenMP constructs, and code transformation to support computing devices including CPUs, GPUs and SIMD units. A full compiler implementation of the latest OpenMP standard for both C/C++ and Fortran would involve a large amount of development efforts spanning multiple years.

Many OpenMP compilers use a high-level IR that is language neutral (or close to neutral) to represent C/C++ and Fortran OpenMP programs. For example, OpenMP support in GNU compiler [8] operates on its high-level and unified IR (named GENERIC and GIMPLE) for C/C++ and Fortran. OpenMP support in IBM XLC compiler [4] also uses its high-level AST-style and unified IR for C/C++ and Fortran for transformation. ROSE's OpenMP implementation [6] operates on the same unified AST representing both C/C++ and Fortran OpenMP input codes. It lowers the AST to generates standard C/C++ or Fortran code with calls to OpenMP runtime functions as its output. OpenMP support in LLVM is an exception so far. The OpenMP compilation for C/C++ are performed within the Clang frontend [1] [3] and for Fortran within the Flang Fortran frontend [2] [9]. There is however effort of extending LLVM IR with intrinsic [11] to perform OpenMP transformation in the LLVM IR, demonstrating the feasibility of OpenMP transformation in a unified and mid-level type of IR.

Our effort to create a standalone and unified OpenMP parser, named as ompparser, is motivated by the facts that 1) the differences in terms of syntax and semantics of OpenMP constructs between C/C++ and Fortran are minor, and 2) current OpenMP compilers develop their own parsers, which represent redundant work. The contribution of our work includes:

- ompparser can be used standalone for static source code analysis, e.g. tools for semantics checking or similarity analysis between C/C++ and Fortran programs.
- Integrating omppaser into an OpenMP compiler implementation can reduce the development efforts. There will be no need to create and maintain two separate parsers for C/C++ and Fortran, or separate parsers for different compilers.
- ompparser provides a complete reference OpenMP grammar in the Backus-Naur Form that formally describes the latest OpenMP language constructs. This will help users understand the rules and restriction of the OpenMP standard, which no longer contains a reference grammar in its recent versions.

In the rest of the paper, we describe the design and interface of the ompparser in Section 2, the ompparser implementation including lexer, grammars and intermediate representation in Section 3, and how it can be used as a standalone tool or to be integrated into a compiler in Section 4. The paper concludes in Section 5.

2 The Design and Interface of ompparser

The ompparser is designed to work as an independent tool or to be integrated into a compiler to parse OpenMP constructs in both C/C++ and Fortran form.

It takes a string of C/C++ pragma processing directive or Fortran comment as input and generates OpenMPIR object as its output representation. OpenMPIR is designed to be in the same form for semantically equivalent C/C++ and Fortran OpenMP constructs. ompparser does not parse the code regions affected by OpenMP constructs. In our current design, it does not parse C/C++ or Fortran expressions or identifiers. ompparser preserves them as plain strings in the OpenMPIR for the compiler to parse. It however provides a callback interface to allow a host compiler to parse expressions and identifiers. ompparser expects the callback to produce compiler-specific IR objects for expressions and identifiers and ompparser attaches those objects as opaque to the OpenMPIR generated for the OpenMP constructs.

The workflow is shown in Fig. 1 and the public interface is shown in Fig. 2. The parseOpenMP method accepts a string of an OpenMP directive to parse, e.g. pragma omp parallel shared (a, b). The optional _langParse parameter can be used by the caller of the parseOpenMP method to pass a language-specific callback function for parseOpenMP to parse language-specific expressions, identifiers, and variables. The _langParse callback should return a pointer to a compiler-specific IR object to ompparser which attaches that object as opaque object to the OpenMPIR. If no _langParse callback is provided when the parseOpenMP is called, language-specific expressions, identifiers, and variables are attached as literal string to the OpenMPIR object. When ompparser is used by a compiler, the compiler may choose to use the OpenMPIR directly returned by ompparser or translate OpenMPIR to the IR of the compiler. Section 3.3 provides the description of the IR and methods to access the IRs. ompparser can also be used as standalone library, used for source code analysis.

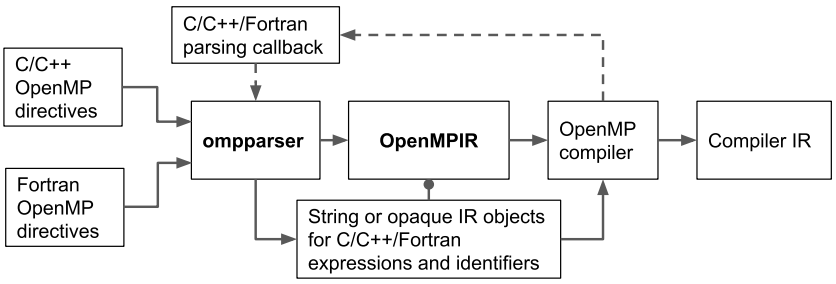


Fig. 1: ompparser integration with an OpenMP compiler

3 Implementation

To help describe the details of our work, we categorize the directives and clauses of the standard based on the complexity of the language constructs.

For directives, we have the following categories:

```
typedef enum OMPLang {
    C, Cplusplus, Fortran,
} OMPLang_t;

//Set the base language for the paser
void OMPSetLang(OMPLang_t lang);

// _input: a string of an input OpenMP directive/commet
// _langParse: a callback function for expression parsing
OpenMPDirective* parseOpenMP(const char* _input,
   void * _langParse(const char*));
```

Fig. 2: ompparser's Interface Functions

- A declare target region can contain multiple function declarations/definitions, which would result in multiple additional declarations. It has no association with the immediate execution of any user code. Declarative directives in the latest 5.0 standard include declare simd, declare target, declare mapper, declare reduction and declare variant.
- An executable directive has immediate executable code associated with it. Most executable directives have a simple structure with a directive name and a clause list.
- Metadirective is a special case of executable directive since the directive-variant parameter of the when and default clause could also be a directive construct.
- Each combined directives is considered as a single directive.

For clauses, we categorize them into three classes:

- clauses with no, one or multiple OpenMP-defined constant parameters, for example, nowait, untied, default in parallel.
- clauses with only a language expression or list as its parameters, e.g. num_threads, private, shared, etc, and
- clauses with one or multiple parameters of OpenMP-defined constants, and then a language expression or list, e.g. map, depend, allocate, reduction, etc.

3.1 Lexer for Tokenizing Keywords, Expressions and Identifiers

The first step of parsing uses a lexer or scanner to tokenize OpenMP keywords, as well as C/C++ or Fortran expressions and identifiers used in the directives and clauses. We use FLEX (Fast Lexical analyzer generator) lexer generator to generate the lexer based on the regular expressions and action rules for the matching tokens. For directives and clauses that require no parameters (class 1 and class 2 mentioned above) as well as OpenMP-defined constants, lexer returns the enum representation of the construct to the parser when the token is matched.

For clauses that have parameters, expression and identifiers (class 3 mentioned above), we use the Flex mechanism for conditionally activating rules based on state to process tokens. This feature makes the Flex rules better organized and versatile to deal with the large number of clauses in OpenMP. We take the reduction clause as an example and its Flex rules are shown in Figure 3.

reduction([reduction-modifier,] reduction-identifier : list)

```
1  reduction      { yy_push_state(REDUCTION_STATE);return REDUCTION; }
2  <REDUCTION_STATE>inscan/{blank}*,       { return MODIFIER_INSCAN; }
3  <REDUCTION_STATE>task/{blank}*,         { return MODIFIER_TASK; }
4  <REDUCTION_STATE>default/{blank}*,      { return MODIFIER_DEFAULT; }
5  <REDUCTION_STATE>"("                    { return '('; }
6  <REDUCTION_STATE>")"                    { yy_pop_state(); return ')'; }
7  <REDUCTION_STATE>","                    { return ','; }
8  <REDUCTION_STATE>":"                    { yy_push_state(EXPR_STATE);
       return ':'; }
9  <REDUCTION_STATE>"+"                    { return '+'; }
10  ...
11 <REDUCTION_STATE>min/{blank}*:          { return MIN; }
12 <REDUCTION_STATE>max/{blank}*:          { return MAX; }
13 <REDUCTION_STATE>.                      { yy_push_state(EXPR_STATE);
       current_string = yytext[0]; }
```

Fig. 3: Flex rules for the reduction clause

To recognize expressions, we develop an expression tokenizer to identify individual expressions from a list without the need to fully parse the expression. A string for a list of expressions separated by "," is split into a list of strings of expressions. The same approach is used for handling shape expression or range-specification used in array sections (e.g. c[42][0:6:2][:]) and other places. While processing the string, the expression tokenizer pairs up brackets ("(", ")", "[", "]", etc) used in an expression and ignore other characters within a bracket pair. Using this approach, the expression tokensizer is able to handle all the cases of expressions in C/C++ and Fortran. For one special case, in some clauses we will encounter the form type id;. type refers to the data type, and id refers to the user-defined identifier. In this case, we take the type and id as one expression. Unlike the normal expression, we need to save the space as well. Then C compiler can compile this expression correctly for us.

3.2 Parser for OpenMP Constructs

This section describes the Bison grammar for OpenMP directives. With Flex rules and Bison grammar, a parser is automatically generated. Bison is a look ahead left to right (LALR) parser generator. Compared to Left to Right (LL)

parser generators, an LALR parser parses the text by the flexible production rules, which enables developers to create a grammar for complex OpenMP structures. Besides, LALR parser is much easier than LL parser to construct grammar since it can be left-recursive.

The grammar structures for executable and declarative directives are very similar. Declare mapper is used to explain how the grammar was generated in Bison, since it represents other declarative directives as well as most executable directives. Fig. 4 shows the grammar of declare mapper directive. The enum values of directive name, which is tokenized by the lexer, is used for start of directive grammar. Declare mapper has two directive-related parameters: mapper_identifier and type_var, which are considered as mapper_parameters. Mapper_identifier includes two identifiers: default and user defined identifier. Default identifier uses DEFAULT as terminal symbol, and user defined identifier uses EXPR_STRING as terminal symbol. For type_var, the reason why we use EXPR_STRING as token was introduced in last section. Mapper clauses are concluded into a declare_mapper_clause_optseq. All attributes must be stored into OpenMP intermediate representation (IR) by putting an action in the middle of a rule (Mid-Rule Actions in Bison's term).

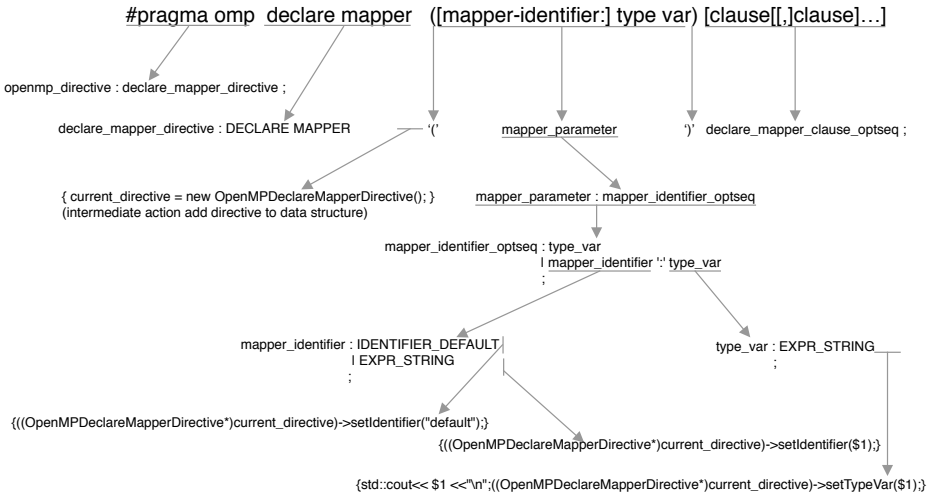


Fig. 4: The grammar for declare mapper directive

The grammar structure for most clauses are very similar. Schedule clause is used as an example to explain how grammar was generated for OpenMP clause. Fig. 5 shows the grammar of schedule clause. Schedule clause has multiple parameters. In those schedule parameters, kind is the only non-optional parameter. Schedule clause should be added to its directive right after kind parameter since we have all parameters information after rule of kind parameter. The information of two modifiers should be stored separately as the Fig. 5 shows. Since those two

modifiers are optional and different from each other, recursive grammar can not distinguished them. We use two global variables to store them. At last, chunk size is stored as independent attributes.

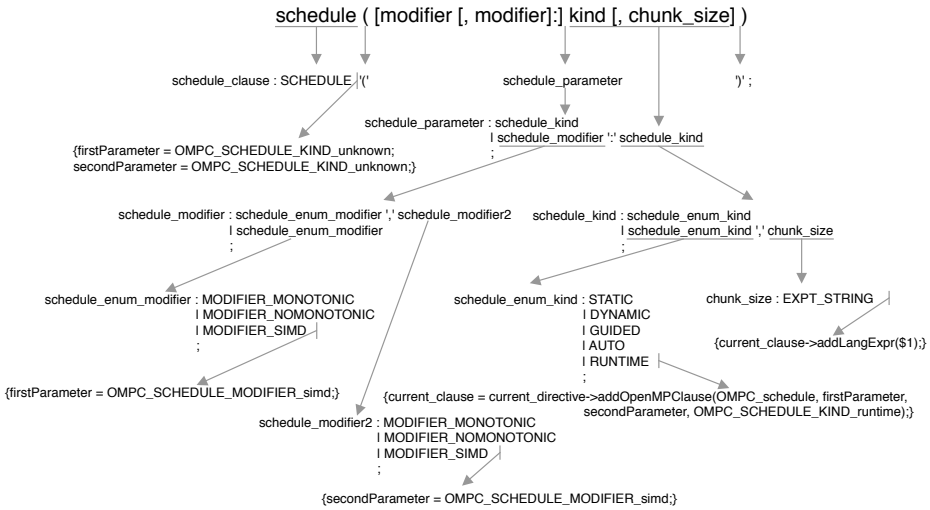


Fig. 5: The grammar for schedule clause

Metadirective is a special directive which contains nested directive and clause structure. Information of nested directives and clauses should be stored appropriately. As Fig. 6 shows, Metadirective has two kinds of clauses. The when clause is similar to other clauses, but it has a context selector and directive variant which may be a directive on its own. A context selector has two parts: specification and trait set selector. Especially, when construct is used as trait set selector, Construct directive will be added as current directive. Therefore, we have designed a nested directive structure. To solve this problem, information of directives and clauses should be stored via two global variables. After construct directive finished its parses, clause and directive should switch back to when and metadirective. Directive-variant also has own directive and clause, same method can be used to handle it. Additionally, it may nest with another directives, but grammar will handle nested directive automatically. Default clause only has one parameter – directive variant. Default clause can be parsed in same way of when clause.

For error handling, both syntax and semantic errors will be checked by grammar. Once an error is found, an corresponding error message will be printed. Then the program won't be crashed but a null pointer will be returned to host compiler/tool to indicates the parsing failed. In this project, a single clause can be used by several directives. And each directive maintains its own clause sequence. Each clause will be added to the directive through an intermediate ac-

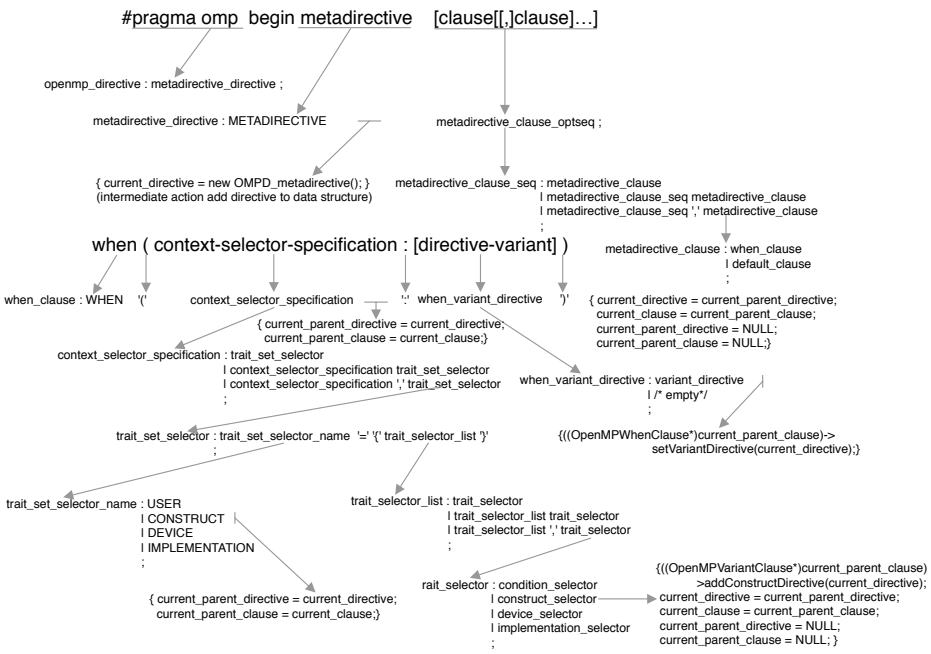


Fig. 6: The grammar for Metadirective

tion. A combined directive is considered as a new directive rather than a nested directive. If a combined directive is used as nested directive, Bison will report a reduce-reduce error.

3.3 OpenMP Intermediate Representation

The intermediate representation of ompparser is an abstract syntax tree for an OpenMP directive. The root node of the tree is the object of the directive and the child nodes are usually the objects for the clauses. Nodes for clauses are stored in a map structure with a clause type as its keys, and a vector of clause objects as its values. For the same kind of clauses that may be used multiple times, e.g. reduction(+:a) and reduction(-:b), the clause objects are stored in the vector of the clause map. With this data structure, searching clauses of a specific kind takes constant time. Searching a specific clause of a specific kind is also fast, since in general we anticipate users would not use the same kind of clause for many times in a directive. Cases where users include many clause in the same directive (e.g. map clause) are uncommon.

There are three methods that are related to how the clause objects are added to directive object – appendOpenMPClause, searchOpenMPClause and normalizeClause. appendOpenMPClause can be used to add a clause after the clause parameters are identified. searchOpenMPClause is used to search clauses of a specific kind and parameters from the OpenMPIR map. normalizeClause can be used to

combine objects for the same kind of clause that also have identical parameters but variable list. For example, objects for two clauses reduction(+:a) and reduction(+:b) can be normalized into one object with two variables, reduction(+:a,b). Clause normalization can be performed alone after a directive is fully parsed or performed while adding a clause to the clause map of a directive in the appendOpenMPClause method. For the later approach, the appendOpenMPClause methods would call searchOpenMPClause methods to retrieve a list of clause objects of the same kind as the one being appended. It then searches to determine whether there's a matching clause to combine with. A new clause is created only if no such clause exist and the reference to this new clause is returned. Otherwise, the existing clause is updated with new information and reference to the clause is returned.

In ompparser, OpenMPIR is implemented with two main C++ classes, the OpenMPDirective class and OpenMPClause class, shown in Figure 7. The OpenMPDirective class can be used to instantiate most OpenMP directives that only have OpenMP-defined clause names. For directives that may have extra parameters, such as declare variant variant_func_id clause1 clause2 ..., the OpenMPDirective class need to be extended to include more fields for those parameters. Similarly directives that allows for user-defined clause names, for instance the requires directive, it needs to be extended to include user-defined clause names. Figure 7 shows the OpenMPDeclareVariantDirective class. For clauses, the OpenMPClause class is used to instantiate OpenMP clauses that have no parameters (class 1 and 2 clauses). For clauses with parameters (class 3), a subclass is needed that includes the fields for the extra parameters of the clause. Figure 7 shows the OpenMPReductionClause class. The OpenMPClause class has a field of vector named as expressions for storing strings of expressions (or list items) specified for a clause. The expressionNodes field is a vector that can be used to store the opaque objects returned by the callback for parsing language expressions.

3.4 Unparsing and Testing

In ompparser, toString and generateDOT methods are provided to unparse the OpenMPIR to its original source code and in a DOT graphic file, respectively. An automated test driver is also implemented. The test driver takes test cases included in source files as its input, creates OpenMPIR, and unparses it to a text output for correctness checking. All the cases in the input file are checked automatically by the test driver and a summary will be printed in the end.

4 Preliminary Results

We are actively developing ompparser to add more OpenMP 5.0 support. Still, we have conducted preliminary evaluation of its current version.

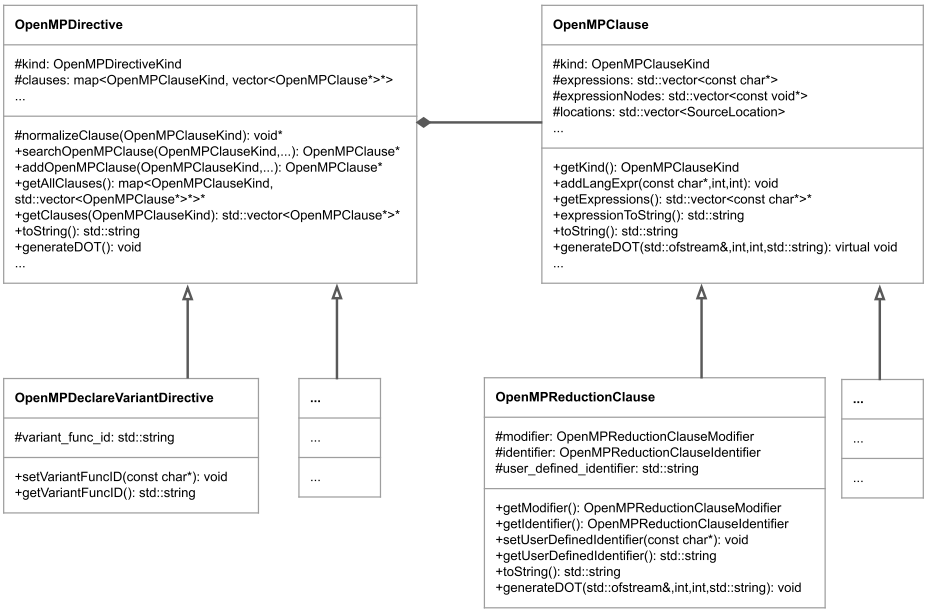


Fig. 7: OpenMPIR architecture

4.1 Used as a Standalone Parser

To evaluate our initial implementation, we use two simple examples in Fig. 8, which are C and Fortran programs that share the same functionality. Ompparser produces the identical OpenMPIR (Fig. 9a) except that the code in Fortran has a second OpenMPIR since the !$omp end parallel is considered as OpenMP code as well (Fig. 9b). Ompparser can merge the OpenMPIR for end with the IR for begin through a normalization step. In the source code, there are two shared clauses. But in the OpenMPIR, they are combined to one after normalization.

The metadirective in the latest OpenMP 5.0 is also supported in ompparser. The example shown in Fig. 10a can switch conditionally between sequential and parallel computing with metadirective. In its OpenMPIR, the parallel directive is attached to default clause as a child node (Fig. 10b).

Ompparser is also able to determine syntax errors existing in the input. For example, the OpenMP 5.0 code #pragma omp parallel if(task: n<3) is provided by user. In if clause, it can have an optional directive-name-modifier, such as task, parallel and so on. But it has to be the same as the directive. In this case, if clause cannot have task as modifier because it belongs to parallel directive. Only parallel modifier is allowed in this particular if clause. ompparser will report the syntax error and return null object to the host, which indicates that the parsing failed.

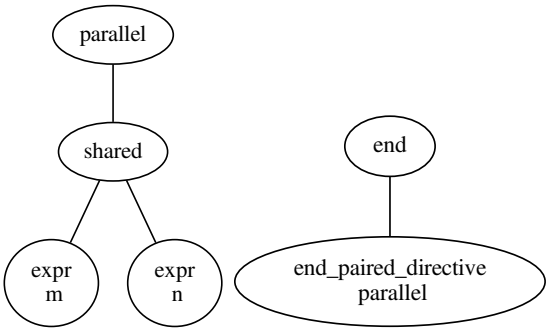
```
1 void foo(int m,int n) {
2 #pragma omp parallel shared(m)
      shared(n)
3     if (omp_get_thread_num() < m)
4       printf("%d\n",n);
5 }
```

(a) C

```
1 subroutine foo(m,n)
2   integer m,n
3 !$omp parallel shared(m) shared(n)
4   if (OMP_GET_THREAD_NUM()<m) then
5     PRINT*, n
6   end if
7 !$omp end parallel
8 end subroutine foo
```

(b) Fortran

Fig. 8: OpenMP source code in C/C++ and Fortran



(a) parallel in C and Fortran (b) end in Fortran

Fig. 9: OpenMPIR for parallel in both C and Fortran

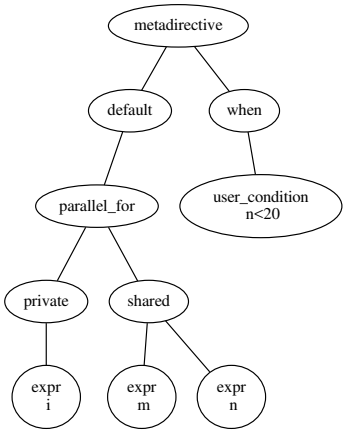
4.2 ROSE Integration

We have also integrated ompparser into the ROSE compiler framework. Developed at LLNL, ROSE [6,7] is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for Fortran 77/95/2003, C, C++, OpenMP, and UPC applications. Internally, ROSE generates a uniform abstract syntax tree (AST) as its intermediate representation (IR) for input codes. Sophisticated compiler analyses, transformations and optimizations are developed on top of the AST and encapsulated as simple function calls, which can be readily leveraged by tool developers.

Fig 11 shows how ompparser is integrated with ROSE. ROSE uses EDG to parse C/C++ codes, and OpenFortranParser (OFP) to parse Fortran codes. However, neither of these two frontends recognizes pragmas or comments for OpenMP constructs. As a result, OpenMP directives are represented as strings in ROSE's AST generated from these two frontends. A separate phase, called OpenMP parsing, is added after the two frontends to parse these OpenMP

```
1  #define n 10000
2  void foo(int* m) {
3  int i;
4  #pragma omp metadirective
5    when(user={condition(n<20)}: )
6    default(parallel for private(i)
           shared(m, n))
7      for(i=0;i<n;i++)
8        goo(m, n);
9  }
```

(a) Source Code in C



(b) OpenMPIR of metadirective

Fig. 10: A metadirective Example

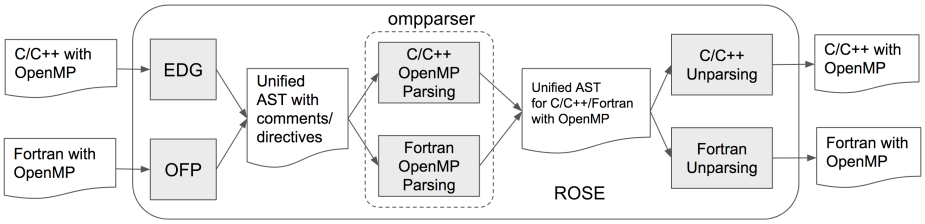


Fig. 11: OpenMP parsing and unparsing in ROSE and ompparser integration

strings. Before using ompparser, two separate parsers were used for parsing C/C++ pragmas and Fortran comments for OpenMP constructs, respectively. The parsing results were first attached to AST as special OpenMP attributes and later translated to dedicated AST nodes representing OpenMP constructs. The ROSE AST has builtin support for OpenMP nodes representing both C/C++ and Fortran, as much as possible, in a uniform way. The unparser is able to parse the same AST into two different output languages (e.g. C or Fortran) as long as their semantics are equivalent.

We replaced the two OpenMP parsers in ROSE with ompparser, as shown in Fig 11. A translator is implemented in ROSE to convert the OpenMPIR produced by omppaser to ROSE's OpenMP AST. The code size and complexity of translator is very similar with the original module in ROSE that generates OpenMP AST. During the conversion, the language expressions are parsed by ROSE's expression parser to produce their AST representation. We have tested the integrated ompparser inside ROSE, using the same examples mentioned in Sec. 4.1. The output of unparsed ROSE AST is identical to its input code when

code in Fig 9 is used. ROSE unparser's support for metadirective is still under development.

5 Conclusion

OpenMP is becoming more and more capable and complicated. It requires a significant amount of efforts for compiler developers to keep their implementations, including parsing, up-to-date. It is not cost effective for every OpenMP compiler to maintain its own parsers while all parsers share the very similar functionality. In this paper, we have presented a standalone and unified OpenMP parser, named ompparser, which can be developed, maintained and used independently. We also took the ROSE compiler framework as an example to demonstrate how to integrate ompparser into other compiler and tools. The initial results show that ompparser's design can support the latest OpenMP 5.0 features such as metadirective. It can also be easily leveraged by an OpenMP compiler. We plan to add more features into ompparser and make it more useful to the OpenMP community.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, and supported by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357. LLNL-CONF-774801. This material is also based upon work supported by the National Science Foundation under Grant No. 1833332 and 1652732.

References

1. OpenMP Support in Clang/LLVM, `https://openmp.llvm.org/`
2. OpenMP Support in Flang/LLVM, `https://github.com/flang-compiler`
3. Antao, S.F., Bataev, A., Jacob, A.C., Bercea, G.T., Eichenberger, A.E., Rokos, G., Martineau, M., Jin, T., Ozen, G., Sura, Z., Chen, T., Sung, H., Bertolli, C., O'Brien, K.: Offloading support for openmp in clang and llvm. In: Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC. pp. 1–11 (2016)
4. Hayashi, A., Shirako, J., Tiotto, E., Ho, R., Sarkar, V.: Exploring compiler optimization opportunities for the openmp 4.x accelerator model on a power8+gpu platform. In: Proceedings of the Third International Workshop on Accelerator Programming Using Directives. pp. 68–78. WACCPD '16, IEEE Press, Piscataway, NJ, USA (2016). https://doi.org/10.1109/WACCPD.2016.7, `https://doi.org/10.1109/WACCPD.2016.7`
5. Leontiadis, I., Tzoumas, G.: OpenMP C Parser (Dec 2001)
6. Liao, C., Quinlan, D.J., Panas, T., de Supinski, B.R.: A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More. pp. 15–28. Springer Berlin Heidelberg (2010)

7. Liao, C., Yan, Y., de Supinski, B.R., Quinlan, D.J., Chapman, B.: Early experiences with the openmp accelerator model. In: OpenMP in the Era of Low Power Devices and Accelerators, pp. 84–98. Springer (2013)
8. Novillo, D.: Openmp and automatic parallelization in gcc. In: In the Proceedings of the GCC Developers (2006)
9. Ozen, G., Atzeni, S., Wolfe, M., Southwell, A., Klimowicz, G.: Openmp gpu offload in flang and llvm. In: LLVM-HPC2018: The Fifth Workshop on the LLVM Compiler Infrastructure in HPC. pp. 1–9 (11 2018)
10. de Supinski, B.R., Scogland, T.R.W., Duran, A., Klemm, M., Bellido, S.M., Olivier, S.L., Terboven, C., Mattson, T.G.: The ongoing evolution of openmp. Proceedings of the IEEE **106**(11), 2004–2019 (2018)
11. Tian, X., Saito, H., Su, E., Gaba, A., Masten, M., Garcia, E.N., Zaks, A.: LLVM framework and IR extensions for parallelization, SIMD vectorization and offloading. In: Third Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2016, Salt Lake City, UT, USA, November 14, 2016. pp. 21–31 (2016). https://doi.org/10.1109/LLVM-HPC.2016.008, `https://doi.org/10.1109/LLVM-HPC.2016.008`