# CLOMP: A Clang-based OpenMP Frontend

Simone Pellegrini
Universit of Innsbruck
spellegrini@dps.uibk.ac.at

April 18, 2013

```
clang::PragmaNamespace* omp = new clang::PragmaNamespace("omp");
pp.AddPragmaHandler(omp);

PragmaHandlerFactory::CreatePragmaHandler<OmpPragmaSection>(
   pp.getIdentifierInfo("section"), tok::eod, "omp"
);
```

Listing 1: Specify a pragma handler in Clang

CLOMP is a framework which allows the definition of user pragmas similarly to EBNF form. The framework takes care of matching pragmas in the input code against the specification given by the user. If the pragma cannot be matched, an error is produced. Otherwise, if the pragma is compatible with the EBNF specification, an annotation node is automatically generated and associated to the corresponding Clang AST to which the pragma is referring.

## 0.1 Registering Pragma Handlers

The implementation of the pragma handling framework is located in the namespace `clomp`. The entry point of the framework is the `BasicPragmaHandler<T>` class, define in `clomp/handler.h`, which is the handler object being registered to the `LLVM/Clang` parser to be invoked when a pragma is encountered in the input file. In order to facilitate the creation of such handler objects, the `frontend::pragma::PragmaHandlerFactory` class is defined.

An example of how a new pragma handler is specified in Listing 1:

First of all `LLVM/Clang` needs a `clang::PragmaNamespace` object to be created which represents the base-name of the pragma. This must be the string which follows the "#pragma" keyword in the input program. In the example we define an handler for `#pragma omp section`, therefore the namespace is defined to be `omp`. After the namespace is created, we register it by adding the handler to the current preprocessor (`pp`) which can be retrieved by the `frontend::ClangCompiler` class (note that the Clang preprocessor shall take ownership of the provided pointer, therefore there is no need to cleanup the memory, this will be done by Clang once the preprocessor is destroyed). The code then generates a specific handler for the "section" keyword using the `PragmaHandlerFactory`. The registration requires the user to provide the keyword for which this handler must be invoked, an EBNF specification (which we will explain later), and the name of the namespace provided as a string. Additionally a template parameter must be provided which represent the class being instantiated to hold the informations contained on the matched pragma (`OmpPragmaSection` in the example).

For new pragmas, the user should define a class in order to process and store the user data contained in the pragma itself. The class `clomp::Pragma` defined in `clomp/handler.h` provide a base class for such purpose. A pragma is defined to store the location of the start and end location and a reference to the Clang node to which the pragma was attached. In C a pragma can refer to two kind of nodes, either declarations (e.g. `clang::FuncDecl`, `clang::VarDecl`) or a generic statement. The methods `isDecl()` or `isStmt()` of the `clomp::Pragma` class shall be utilized to test the type of the target node. The methods `getDecl()` or `getStmt()` can be used to retrieve a pointer to the target node.

## 0.2 Overview of Pragma Matching

The constructor of the class provided to the `CreatePragmaHandler` function is invoked automatically by the framework once a pragma is being matched. The matching process is split into two phases.

**Phase 1** : In Phase 1 the framework tries to match the content of the pragma against the EBNF specification provided by the user. This is implemented using a standard backtracking engine which consumes the input stream until the "end of directive", `clang::tok::eod`, token is encountered. If the matching cannot be performed, an error is produced and the pragma is discarded.

If instead the pragma is correctly matched, an instance of the user provided object type is generated and stored in a list of "pending" (or unmatched) pragmas.

**Phase 2** : The Phase 2 takes care of attaching pragmas to the corresponding statements. Because `LLVM/Clang` processes the pragma before looking at the target statement (and therefore an AST node is not available by the time the pragma is processed), the association must be performed lazily. Because the lack of any context information when a pragma handler is being invoked, the matching is performed solely on the basis of source code locations.

### 0.2.1 Pragma Specification

A pragma specification is provided to the framework as an expression built using C++ operators in a way which resembles the EBNF form. The specification expression composes a tree which is implemented following the composite design pattern for which the `clomp::node` class (defined in `clomp/matcher.h` is the abstract base. The leaves of the generated tree are lexer tokens. A pragma specification can be built using the following 4 operators:

`t1 >> t2`**:** Binary operator which represents the concept of *"concatenation"*, it matches the input stream the next two tokens are respectively `t1` followed by `t2`;

`t1 | t2`**:** Binary operator which represents the concept of *"choice"*, it matches the input stream if the next token is either token `t1` or token `t2`;

`!t`**:** Unary operator which represents the concept of *"option"*, it matches the input stream if the token `t` is present 0 or 1 times;

`*t`**:** Unary operator which represents the concept of *"repetition"*, it matches the input stream if the token `t` is present 0 or N times;

In each case a token `t` can be either a lexer token (leaf node of the expression) or an expression tree. Using these operators is possible to define, for example, the `for` clause of an OpenMP for (the full code can be found in `clomp/omp/pragma.cpp`, see Listing 2.

```
1   auto kind =
2      Tok<clang::tok::kw_static>() | kwd("dynamic") | kwd("guided") |
3      kwd("auto") | kwd("runtime");
4
5   auto op = tok::plus | tok::minus | tok::star | tok::amp |
6           tok::pipe | tok::caret | tok::ampamp | tok::pipepipe;
7
8   auto var_list = var >> *(comma >> var);
9
10  auto reduction_clause = kwd("reduction") >>
11     tok::l_paren >> op >> tok::colon >> var_list >> tok::r_paren;
12
13  auto for_clause =
14        reduction_clause
15     |  (kwd("schedule") >> tok::l_paren >> kind >>
16        !( tok::comma >> expr ) >> tok::r_paren)
17     |  (kwd("collapse") >> tok::l_paren >> expr >> tok::r_paren)
18     |   kwd("ordered") | kwd("nowait")
19     ;
```

Listing 2: OpenMP for clause definition

As already stated, the leaf nodes of the expression are lexer tokens which are imported from `LLVM/Clang` token definitions (see in the `clang/Basic/TokenKinds.def`) and made available under the `clomp::tok` namespace. Beside to the preprocessor tokens of `LLVM/Clang` we define several new leaf nodes for the purpose of simplifying the specification of new pragmas.

**kwd( "str_lit" ):** the matcher expect to encounter an identifier which is exactly the literal provided as argument. Note that because we use the C lexer, keywords which are recognized to be reserved words in the C/C++ language cannot be matched in this way. In such cases the name of the `LLVM/Clang` token must be used, for example keyword `kwd("static")` is not allowed as this is a reserved keyword in C. The `clang::tok::kw<static>` must instead be used.

**expr:** This placeholder matches any C/C++ expression. Indeed, usually pragmas may contain expressions. One limitation of the `LLVM/Clang` pragma matching mechanism is that the C parser is not made available to pragma handlers. However this is more of a Clang design limitation rather then capability. In order to overcome this problem CLOMP works on a patched version of the `LLVM/Clang` compiler. The patch makes the engine for pragma matching of CLOMP be able to retrieve the instance of the parser (`clang::Parser`) object. This allows CLOMP to invoke the parser even during the processing of pragmas (which is not allowed by relying solely on `LLVM/Clang` API. This means that complex C/C++ expressions can be used in the pragma specifications and semantics check can be automatically performed on those (e.g. use of undeclared variable).

**numeric_constant:** Another useful placeholder which specifies that the token to match must be any valid numeric constant.

**var:** Makes sure that the matched token is a valid variable. This not only assures that the token is a valid C identifier, but also that the variable has been declared. Use of undeclared variable will be captures as an error by the preprocessor.

### 0.2.2 Pragma Matching

The matcher uses the expression and invoke the `match()` virtual function which has a different implementation for every type of connector. In this way we are able to implement a sort of backtracking engine which tries to consume the input stream until a match is obtained. If not, an error message is automatically produced listing the alternatives which the matcher was expecting at the specific location and it couldn't find instead. The error message is returned back to the user using the `LLVM/Clang` diagnostic engine showing the precise code location where the error occurred.

Matching the structure of the pragma is just a part of the whole story. Indeed, the user may be interested not only to know that a statement has associated a particular type of pragma, but, most likely, it may be interested to its content. Usually, the information contained in the pragma are mostly syntactic sugar which, once the pragma has been matched, loose any function. Because the framework cannot decide by itself what is interesting and not for the user to be stored, we define two additional operators which allows the user to specify what should be extracted from a particular pragma once is matched.

**`["key"]`:** At any point of the pragma specification the `[]` operator can be used to force the framework to store all the tokens matched by the node to which the operator is applied. Informations are store into a multimap where the value of the key is the string value provided as argument of the `[]` operator. For example, `(var >> *(comma >> var))["VARS"]` stores all matched tokens into a map having `"VARS"` as a key and the list of matched tokens as value. If the following input is encountered: `a, b, c` the resulting map will be of the form: `("VARS" -> { a, b, c, "," })`.

**~:** As seen before, sometimes we want to be able to *exclude* specific type of tokens to be mapped to the resulting result. This is the purpose of the ~ operator which forces any token mapped by the addressed expression to be removed from the mapping. Therefore, by changing our specification for variable lists to: `(var >> *( comma >> var))["VARS"]`, the resulting multimap will be the following: `("VARS" -> {a, b, c})`.

The result of the matcher is therefore an object of type `clomp::MatchMap`, which is defined in `clomp/matcher.h`. Each key is matched to a list of objects which can be either a pointer to a string (used when the pragma matches string literals for example) or a pointer to a generic `clang::Stmt*`. This is the case when the matched token is a C expression or for example a variable identifier. It is worth noting that, by default, keywords nodes are inserted into the matching map without the need for the user to explicitly specify the mapping. A `kwd("auto")` for example will create an entry in the map whose key is `"auto"` and the value is an empty list. A recurring use case is represented in the code snippet of Listing 3.

```
1 auto var_list = var >> *(~comma >> var);
2
3 auto private_clause =
4    kwd("private") >> tok::l_paren >> var_list["private"] >> tok::r_paren;
5
6 auto for =
7    kwd("for") >> !private_clause;
```

Listing 3: Example of pragma definition with CLOMP

Where the `"private"` keyword is utilized to capture the list of variables associated to the clause. Given the following pragma `#pragma omp for private(a,b)` the generated matching map will be the follow: `"private" -> { a, b }; "for" -> { }`

The generated map is forwarded to the object constructor registered through the `PragmaHandlerFactory`. The pragma object is constructed iff the matcher is able to completely match the user specification against the input code. The user object is responsible to analyze the matching map returned by the pragma matcher and do the necessary operations. An explanatory example of how OpenMP pragmas are processed and stored is in the `clomp/omp/pragma.cpp` file.

### 0.2.3 AST Node Mapper

The second phase of the pragma framework takes care of associating, or mapping, generated pragma objects to the AST nodes to which pragmas refer. The framework does this operation in a way which is transparent to the user. The code which takes care of this aspect is in the `clomp/sema.h` and `clomp/sema.cpp` files.

Two solutions are possible in order to connect pragmas with statements; one solution would be to traverse the entire AST after it has been generated. However this solution requires an expensive traverse of the input program AST and this could be inefficient for big codes, especially since the pragma/statement ratio is usually very small.

The way the `LLVM/Clang` compiler builds the AST is by invoking "actions" provided by the `clang::Sema` class which reduces the tokens currently available on the parser stack and generates the corresponding AST node, and append it to the program AST. This is the best location for implementing our matching algorithm for pragmas. Indeed we can keep the list of processed pragmas and every time a new statement is being created by the `clang::Sema` class we check, based on the location, whether any of the pending pragmas refer to the newly generated statement. With this solution the overhead produced by the matching is minimal as our lookup procedure is local to the code section defining pragmas. For code segments containing no pragma we do not pay any overhead.

Fortunately, when the first version of CLOMP was developed, `LLVM/Clang` interfaces allowed for the class `clang::Sema` to be extended. As a matter of fact all the action methods were virtual allowing for extending its behaviour. Starting with `LLVM/Clang` 2.9 the interfaces had a dramatical change the `LLVM/Clang` developers removed the possibility to customize the actions of the `clang::Sema` class. In order to overcome this limitation imposed by the `LLVM/Clang` developer we patch the clang code making virtual the functions of the `clang::Sema` class for which we need to redefine the behaviour.

As stated before, the matching algorithm works with the only context information available at this stage, i.e. locations. The check for pragma matching is performed for few selected node types, i.e. `ActOnCompound-Stmt`, `ActOnIfStmt`,`ActOnForStmt`, `ActOnStartOfFunctionDef`, `ActOnFinishFunctionBody`, `ActOnDecla-`

rator, ActOnTagFinishDefinition. The algorithm keeps a list of pending pragmas ordered by their locations. Once one of the statements is reduced, we check which pragmas are within the range of the statement. If none, we check whether any of the pending pragmas are located right before the statement. In that case the pragma gets associated to that statement and we remove the pragma from the list of pending pragmas. If pragmas are located within the range of the statement we iterate through the children and match accordingly to the positions.

### 0.2.4 Detached Pragmas

One tricky aspect of the entire algorithm is how we deal with pragmas which are not meant to be attached to a statement. An example is the OpenMP `#pragma omp barrier`. Indeed this pragma is not meant to be associated to a statement, for example the code in Listing 4 is a valid OpenMP input code:

```
1  {
2      ...
3      #pragma omp barrier
4  }
```

Listing 4: An example of a positional pragma

Because the `LLVM/Clang` compiler doesn't represent pragmas in the AST we need a way to easily map the location of a pragma to a node in the AST. So that when the AST is traversed for the IR conversion, we can handle the pragma. However, when a pragma does not not refer to a statement the matching algorithm creates an empty statement (a `no-op`, `;`) and transform the `LLVM/Clang` AST by inserting the no-op at the correct location. We then map the pending pragma to the generated statement. In order to check whether the statement associated to a pragma is generated by CLOMP it is necessary to query the statement for its location. If the returned `clang::SourceLocation` object is not valid, then it means this statements has been introduced by the matching algorithm.

### 0.2.5 Traverse (and Filter) Pragmas

Each `clomp::TranslationUnit` object has a reference to a list of Pragmas object being generated by a file (`clang::ASTContext`). Therefore given a translation unit, we can retrieve the list of pragmas in that unit. An example is depicted in Listing 5.

```
1  Program p;
2  TranslationUnit& tu = p.addTranslationUnit("input.c");
3  for(const PragmaPtr& cur : tu.getPragmaList()) { LOG(INFO) << *cur; }
```

Listing 5: Example of how to use CLOMP

However, it is usually more useful to retrieve the complete list of pragmas across translation units. This can be done through the `Program` class which offers two methods `pragmas_begin()` and `pragmas_end()` which returns an iterator through all pragmas in the input program. Additionally a filter can be passed so that the user obtains only pragmas of a certain category. For example iterating through the all `"omp"` pragmas can be easily done as follows:

```
1   auto pragmaMarkFilter = [](const pragma::Pragma& curr) {
2      return curr.getType() == "omp";
3   };
4   for(Program::PragmaIterator pIt = pragmas_begin(pragmaMarkFilter),
5                               pEnd = pragmas_end(); pIt != pEnd; ++pIt)
6   { ... }
```

Listing 6: Iterate through pragmas in multiple TUs