OpenMP Support in Clang

Design and Implementation Proposal

Mahesha S, Prakash Raghavendra, Dibyendu Das

Table of Contents

1 Introduction

1.1 Purpose

This document describes design and implementation proposal for supporting OpenMP in Clang. It also describes the current status of design and implementation of the same with a plan to put the basic infrastructure code being implemented till date for review.

Authors assume that readers have basic understanding of:

1. OpenMP principles and language constructs
2. Knowledge of basic internals of Clang compiler front-end

1.2 Goal

The goal of this proposal is to provide a neat and solid support for OpenMP in Clang compiler front-end.

1.3 Scope

Scope of this document at present is limited to design and implementation proposal for syntax analysis, semantic analysis, and AST representation of OpenMP constructs in Clang compiler front-end. Thus, all other issues related to OpenMP supports like AST lowering, runtime library, ABI, etc are not covered in this document.

We would like to inform that we are currently keeping track of all the discussions/proposals that are currently put by LLVM community to support OpenMP in LLVM [4], [5], [6]. However, a consensus has not yet been reached, and so we're skipping the topic of OpenMP AST lowering from our current proposal. However, in our design and implementation, we will make sure that the OpenMP representing ASTs will store all the relevant information, and they are flexible enough to any kind of lowering decisions – either, to completely lower OpenMP constructs to OpenMP runtime calls in Clang compiler front-end itself, or, to postpone the lowering part to LLVM back-end.

Also, we would like to know about any other on-going effort to support OpenMP in Clang.

Our proposal is based on the latest published OpenMP specification, which is version 3.1 at the time of writing. However, the design approach we employed is

general enough to allow easy adaptation for future versions of OpenMP standard.

In the next section (section 2), we summarize the general implementation of OpenMP in a typical compiler. In section 3, we briefly describe the general tasks involved to support OpenMP in Clang. Section 4 briefly discusses our design and implementation proposal for implementing OpenMP in Clang. Section 5 briefly describes the proposed AST design to represent OpenMP constructs in Clang. In the section 6, we brief about the current status of our implementation which is followed by conclusions and acknowledgement in section 7 and section 8 respectively, with references in section 9.

2 Brief Summary of OpenMP

OpenMP is comprised of four components (as of version 3.1):

- Directives (+ Clauses)
- Internal Control Variables
- Runtime Library Routines
- Environment Variables

All OpenMP directives in C/C++ are specified with **#pragma** preprocessing directive and have the following format:

#pragma omp *directive-name [clause [[,] clause]…]*.

Quick summary of the OpenMP 3.1 API C/C++ syntax and semantics is available at [1], and detailed specification is available at [2].

OpenMP support in a compiler, in general, involves following steps. However, note that there may be additional steps involved depending on how the compiler is implemented and how the program being compiled is internally represented.

1. Parsing of OpenMP directive statements
2. Semantic analysis of the parsed OpenMP directive statements
3. Internal representation of parsed OpenMP directive statements
4. Outlining of OpenMP parallel regions as separate functions
5. Lowering of OpenMP constructs to OpenMP runtime library function calls

A typical compiler, in general can implement the above five steps in one of the following two ways.

1. All the above steps from 1 to 5 are implemented in the compiler front-end it-self.
2. Steps from 1 to 3 are implemented in compiler front-end and steps from 4 to 5 are implemented in compiler back-end.

In our proposal, we have planned to implement steps 1, 2 and 3 in Clang compiler front-end, and steps 4 and 5 are deferred until a consensus is reached on LLVM side.

3 General OpenMP Support Tasks in Clang

3.1 Clang Driver Support

Following are general OpenMP support tasks required by Clang driver.

1. The Clang driver understands the OpenMP enablement option being passed by the user. The option for the same is “-fopenmp”.

2. Upon seeing the option “-fopenmp” in the option list, it enables the OpenMP macro “_OPENMP”, and adds the linker option to link to the OpenMP runtime library.

3. Finally, the driver passes the option “-fopenmp” to other OpenMP handler components like Clang compiler proper.

3.2 Clang Compiler Proper Support

Following are general OpenMP support tasks required by Clang compiler proper. While parsing, when the Clang compiler proper encounter with the OpenMP pragma directive statement, it checks, if the OpenMP support is enabled through the option “-fopenmp”. It does one of the following two tasks based on whether the option “-fopenmp” is enabled or not.

1. When the “-fopenmp” is enabled, it does the syntax and semantic analysis of OpenMP directive statement encountered. In case of any syntax and semantic errors, it reports the relevant error messages and stops further compilation. Upon successful syntax and semantic analysis, it builds ASTs. Finally, CodeGen component of Clang will perform the AST lowering task.

2. When the “-fopenmp” is not enabled, it throws a warning saying that OpenMP directive statement will be ignored, and simply eats-up (ignores) the OpenMP pragma directive statement line, and proceeds with the sequential compilation as usual. In this case, any call to OpenMP runtime library function within the code will results in linker error (undefined reference) as the OpenMP runtime library will not be linked in the absence of “-fopenmp” option.

4 Implementation of OpenMP in Clang: A proposal

4.1 Clang Driver Implementation

Clang driver implementation for the OpenMP support is quite straightforward, and it involves the support for "-fopenmp" option as mentioned in the section 3.1.

4.2 Clang Compiler Proper Implementation

Following is the brief design and implementation proposal of OpenMP support in Clang compiler proper.

4.2.1 Summary

- Our implementation will strictly adhere to design and implementation philosophy behind Clang.
- We extend the existing Clang components like Parser, Sema, AST, etc in order to handle OpenMP in Clang.
- We add a new component called OmpPragmaHandler to Clang. The main job of OmpPragmaHandler is to assist other Clang components like Parser, Sema, etc while handling OpenMP constructs.
- Tokens are introduced to represent only OpenMP pragma directives, and these tokens are inserted by OmpPragmaHandler to token streams, upon seeing the OpenMP pragma directive statements. Lexer is completely not aware of these tokens. Additionally, OpenMP pragma clause names are parsed based on the context with string comparison approach. This strategy avoids conflicts with "Identifier" tokens that would happen when we introduce separate tokens for each and every OpenMP construct names.
- As in the case of parsing any other C/C++ constructs, Parser starts parsing the OpenMP directive statements when it encounters with OpenMP directive tokens, which are inserted by OmpPragmaHandler. Once the parsing is done, Parser calls Semantic Analyzer to perform semantic analysis, and to build AST tree which represents parsed OpenMP directive statement. Finally, CodeGen component of Clang is called to lower OpenMP ASTs.

4.2.2 Details

- A new library called “clangOmp.a” will be added to Clang, which assists other libraries in Clang like “clangParse.a”, “clangSema.a”, etc while processing OpenMP constructs. That is, “clangParse.a”, “clangSema.a”, etc will act as clients to “clangOmp.a”, with “clangOmp.a” it-self be a client to “clangLex.a” and “clangBasic.a”. Hence, the link order used to create Clang compiler proper looks as below.

  “…… clangParse.a clangSema.a …….. clangAST.a clangOmp.a clangLex.a clangBasic.a”

- Basically, “clangOmp.a” defines and implements a class, called “class PragmaOmpHandler” which interfaces with other classes like “class Parser”, “class Sema”, etc. Both Parser and Sema objects hold a referenced PragmaOmpHandler object in order to get required assistance from it.

- Clang Preprocessor is responsible for registering all the OpenMP pragma directives, and further processing them when OpenMP pragma directives statements are encountered. Clang Parser is responsible for all syntax analysis and syntax errors checking and reporting (if any). Clang Semantic Analyzer is responsible for all semantic analysis and semantic errors checking and reporting (if any). All the ASTs classes are defined and implemented within Clang AST manager component (clangAST.a) with all the additional supporting implementation for AST visiting, AST writing, AST reading, etc. Finally, CodeGen component of Clang (clangCodeGen.a) implements all the AST lowering routines.

- During the initialization of Clang compiler proper before processing a translation unit, a PragmaOmpHandler object is constructed before both Parser and Sema objects are constructed. Upon creation of PragmaOmpHandler, it is made to initialize all its members, and asked to call Preprocessor to get register all the OpenMP pragma directive’s names. Later, both Parser and Sema objects are made to reference PragmaOmpHandler object when they are constructed.

- During the parsing of a translation unit, Clang Lexer, upon lexing the token “#”, calls Preprocessor to handle possibly encountered pragma statement. Clang Preprocessor, after realizing that the encountered pragma is an

OpenMP pragma statement, it calls PragmaOmpHandler to handle it. The PragmaOmpHandler, upon called to handle OpenMP pragma statement, does one of the following two tasks depending on whether the "-fopenmp" option is enabled or not.

1. When the "-fopenmp" option is enabled, it inserts a token which represents the encountered OpenMP pragma statement into token stream so that parser can recognize it and can parse the respective OpenMP pragma directive statement. Parser will later take care of parsing the encountered OpenMP pragma directive statement.

2. When the "-fopenmp" option is not enabled, it simply eats-up (ignores) the OpenMP pragma directive statement line, so that, Parser can proceed with the sequential compilation as usual. However, in this case, a warning message is emitted saying that the current OpenMP directive statement will be ignored. Note that Clang emits only one warning message per translation unit irrespective of the number of OpenMP constructs encountered within the translation unit being parsed. This avoids message bloating on the output device.

- Once the parsing, semantic analysis, and AST creation is completed, Clang CodeGen component is called for lowering ASTs.

5 OpenMP AST Representation in Clang

5.1 An Example AST Representation

Figure 1 shows an AST representation for OpenMP directive statement by considering the "parallel" directive statement as an example.

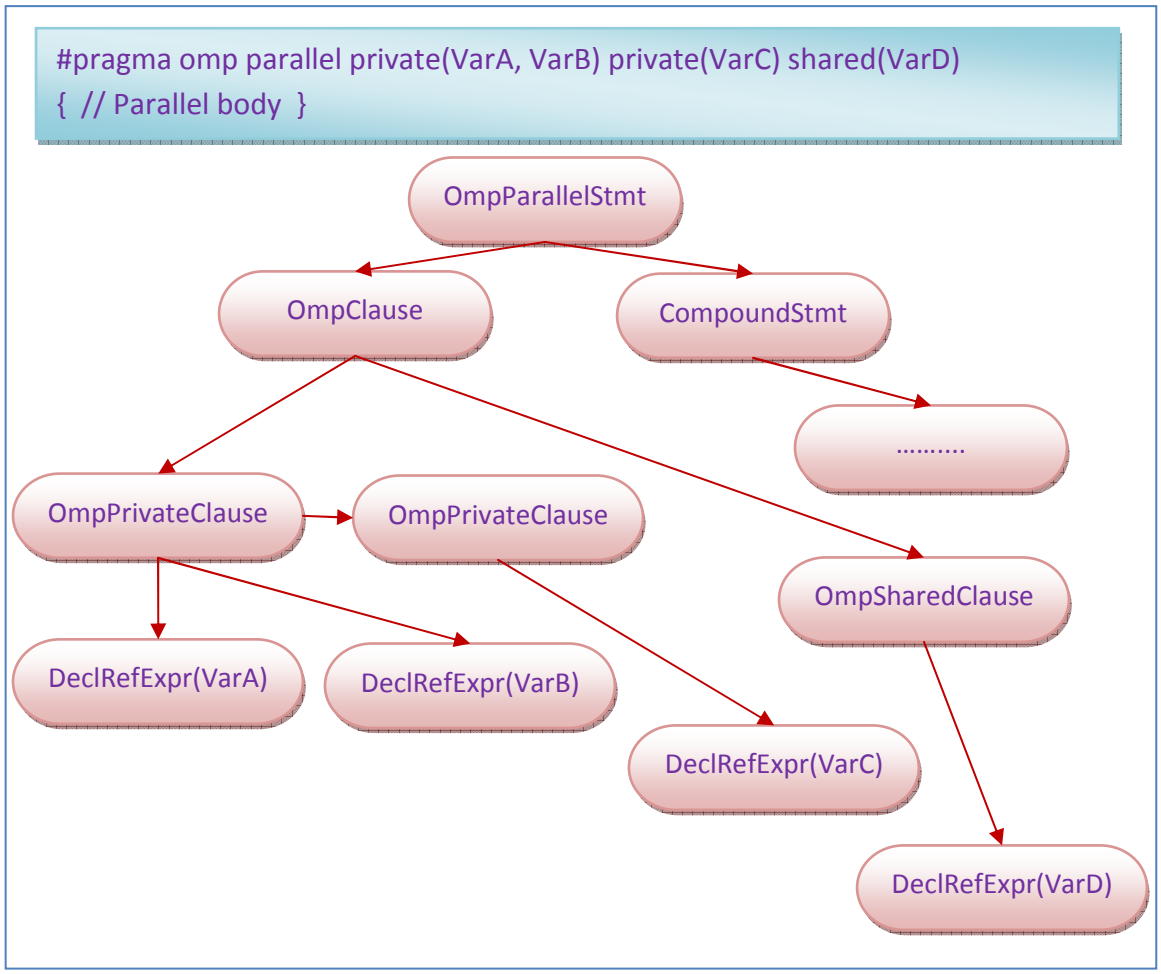

Figure 1

5.2 OpenMP AST Implementation in Clang

Following is the brief summary about AST implementation in Clang.

- All the AST classes which represent OpenMP directives or clauses publicly derive from the statement AST base class namely "class Stmt".
- There will be a separate AST class implementation for each OpenMP directive and clause.

- There will be an additional AST class node implementation, called “class OmpClause”. OmpClause AST node holds other AST clause nodes as its children, which represent the different OpenMP clauses appeared in an OpenMP directive statement as shown in the figure 1. However, note that the same clauses which appear more than once in an OpenMP directive statement are represented as a linked list as shown in the figure 1 for the “private” clause. This representation makes tree traversal easier for few particular semantic checks.

- Variables in the list clauses like “private” clause are represented as “DeclRefExpr” node.

- ASTs are designed and implemented in such a way that *no* OpenMP information is lost including source location information.

Note: More details will be given about OpenMP AST implementation when the code is put for review.

6 Current Status of Implementation

6.1 Clang Driver Support Status

6.1.1 Completed

- The option "-fopenmp" is implemented. Clang driver now understands the above option, and pass it onto the Clang compiler proper.

6.1.2 Yet to be Completed

- To enable "_OPENMP" macro, and to add the linker option to link to the OpenMP runtime library when the driver recognizes the option "-fopenmp".

- To add any other missing support.

6.2 Clang Compiler Proper Support Status

6.2.1 Completed

- A new library, called "clangOmp.a" is added within Clang, and linked it with other Clang libraries as described in the section 4.2. Within this library, an OpenMP pragma handler class, called "class PragmaOmpHandler" is defined and implemented, and this Clang component is made to interface with other Clang components like Parser, Sema, etc as described in section 4.2.

- OpenMP pragmas registration with Clang Preprocessor is implemented.

- AST classes for *all* the OpenMP constructs except for "critical" construct are defined and implemented.

- Skeleton routines (with few partially implemented) for parsing, semantic analysis are introduced.

- Following two things are happening currently with respect to the handling of OpenMP constructs in Clang.

    1. When the user does not pass the "–fopenmp" flag, Clang warns the user that the OpenMP directives will be ignored, then eats-up/discards the OpenMP directive statements.

    2. When the user does pass the "–fopenmp" flag, Clang Parser is made to understand the OpenMP directive tokens, and respective parsing routines are called to parse the OpenMP constructs. But, these parsing routines, currently,

again just eats-up the OpenMP directive statements without actually parsing the OpenMP directive statements.

6.2.2 Yet to be Completed

- We are just in the beginning, with the basic support for infrastructure being laid down.
- Major support for parsing, semantic analysis, AST visiting, AST writing, AST reading, AST lowering, testing, etc is required.
- Decision about runtime library support to be taken. To discuss, if we can go with GCC libgomp library, in case, there won't be any licensing issues in using it. Or, to discuss about implementing altogether a new OpenMP runtime library in Clang/LLVM infrastructure.

7 Conclusions

In this document, we briefly descried our proposal to support OpenMP in Clang with brief description about the status of our current implementation. As mentioned in the section 6, we have currently laid down the basic infrastructure to support OpenMP in Clang as per our design that we proposed in this document.

Further, we are looking forward for your constructive feedback.

In particular, we would like to know about the Clang community opinions about further steps to support OpenMP in *Clang* based on current proposal and its implementation status as mentioned in the section 6. And, we want to discuss next steps for code review, submitting patches, etc, once all the design reviews are taken care. We would also like to have discussions about OpenMP AST lowering, once the consensus is reached on supporting OpenMP in LLVM side.

8 Acknowledgement

Authors would like to acknowledge valuable advice provided by Hal Finkel [4], Prashantha Rao, and Vishwanath Prasad while preparing this proposal document and while prototyping the same.

9 References

[1] OpenMP 3.1 API C/C++ Syntax Quick Reference Card. Available at: http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf

[2] “OpenMP Application Program Interface”, Version 3.1, July 2011. Available at: http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[3] "Clang" CFE Internals Manual. Available at: http://clang.llvm.org/docs/InternalsManual.html

[4] Hal Finkel, [LLVMdev] [RFC] Parallelization metadata and intrinsics in LLVM

[5] Sanjay Das, [LLVMdev] [RFC] Progress towards OpenMP support

[6] Alexey Bataev, Andrey Bokhanko, “OpenMP Representation in LLVM IR: Design Proposal”, [LLVMdev] [RFC] OpenMP Representation in LLVM IR