OpenMP Representation in LLVM IR

Design Proposal

Alexey Bataev, Andrey Bokhanko

Contents

Introduction

This document describes design proposal for OpenMP representation in LLVM IR.

Authors assume that readers have basic understanding of:

1) OpenMP principles and language constructs
2) General design of LLVM compiler system

Goals

Our end goal is to provide a simple, complete and extensible support for OpenMP representation in LLVM IR.

We aim to extend LLVM IR as little as possible, preferably without adding any new types / language constructs at all.

Also, we'd like to keep opportunities for optimization of parallel code as widely opened as possible – obviously, within the boundaries of preserving correct semantics of a user program.

And finally, in the spirit of OpenMP Specification ([OpenMP11]), LLVM IR based compilers should be able to easily skip IR extensions related to OpenMP support and still generate correct, albeit sequential, code.

Function Outlining

Eventually, parallel regions should be put into separate routines, a process usually called "function outlining" or "procedurization". This can happen as early as in front-end, and as late as right before code generation.

As can be seen in the following sections, the IR extension we propose doesn't involve explicit procedurization. Thus, we assume that function outlining should happen somewhere in the LLVM back-end, and usually this should be aligned with how chosen OpenMP runtime library works and what it expects. This is a deliberate decision on our part. We believe it provides the following benefits (when compared with designs involving procedurization done in a front-end):

1) Function outlining doesn't depend on source language; thus, it can be implemented once and used with any front-ends.

2) Optimizations are usually restricted by a single function boundary. If procedurization is done in a front-end, this effectively kills any optimizations – as simple ones as loop invariant code motion. Refer to [Tian2005] for more information on why this is important for efficient optimization of OpenMP programs.

   It should be stressed, though, that in order to preserve correct semantics of a user program, optimizations should be made thread-aware (which, to the best of our knowledge, is not the case with LLVM optimizations).

   By "thread-aware" we mean an optimization that performs legal transformations on parallel programs. Refer to [Novillo00] for more information on correctness of optimizations for parallel programs.

Given that LLVM optimizations are currently not thread-aware, initial implementation should call procedurization pass right at the start of the optimizer. In the future, this pass can be moved further and further down the optimizer, as more optimizations will become thread-aware.

Essentially, this initial implementation is not that different from implementations with procedurization done in front-end; however, it keeps window of possibility opened.

Scope

Scope of this design is OpenMP *representation in LLVM IR*. Thus, all other issues related to OpenMP support, like runtime library, ABI, etc are not covered.

We also included a set of requirements for front-ends and back-ends, which establish mutual expectations and is an important addition to the design.

Our proposal is based on the latest published OpenMP specification, which is version 3.1 ([OpenMP31]) at the time of writing. However, the design approach we employed is general enough to allow easy adaptation for future versions of OpenMP standard.

Front-End/Back-End Contract

While not a part of OpenMP representation design *per se*, the following pre- and post-conditions help to establish a set of mutual expectation between front-ends (that generate LLVM IR with OpenMP support) and back-ends (that consume it). Without this kind of a "contract" ([Meyer92]), a back-end has to verify too much and basically repeat the work already done by a front-end.

While it is possible to generate LLVM IR that violates these conditions, we consider it to be non-conformant. Obviously, conformance can be verified, if necessary.

Pre- and Post-conditions for Front-Ends

1) Guarantee correct semantics of directives and clauses. This includes nomenclature and number of clauses in directives, correct nesting of directives, values of clauses, etc.

For example, front-ends should guarantee that *omp single* directive has at most one *private*, *firstprivate*, *copyprivate* and *nowait* clause, and nothing else; *omp section* directive is nested inside *omp sections* directive; the only possible value of *private* clause is a list of variables available at the point where this clause is present; etc.

2) Guarantee correct semantics of statements / structured blocks following directives.

As an example, it should be guaranteed that only for-loops follow *omp parallel for* directive.

3) Set _OPENMP macro. This is required in section 2.2 of [OpenMP11].

4) Support a command-line option that enables/disables OpenMP support. This is required in Chapter 2 of [OpenMP11].

If a user program violates OpenMP semantics and thus, makes compliance with first two conditions impossible, a front-end should report a meaningful error message and stop compilation.

It should be noted that front-ends are not required to guarantee full conformance of generated LLVM IR to OpenMP specification. As an example, the specification deems programs that branch into or out of *parallel* regions to be non-conformant. We believe that verification of such properties is too complex for most front-ends. This matches well with what is written in the specification itself: "compliant implementations are not required to check for code sequences that cause a program to be classified as non-conforming" ([OpenMP11], Section 1.1).

Pre- and Post-conditions for Back-Ends

1) Intrinsics, builtins, library routines and language implementation should be thread-safe. This is equally applicable to front-ends and is required in Section 1.5 of [OpenMP11].

2) Optimizations should be thread-aware.

3) Support a command-line option that enables/disables OpenMP support. This is required in Chapter 2 of [OpenMP11].

4) Rely on post-conditions guaranteed by front-end, and nothing else.

Conditions 2)-5) are only applicable to optimizations working on IR *before* procedurization.

As noted in the previous section, LLVM IR is not guaranteed to be fully conformant with OpenMP specification. Back-ends should be able to compile cleanly non-compliant programs, but no promises are made on correct behavior of resulting machine code.

Elements of OpenMP

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

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

Internal Control Variables, Runtime Library Routines and Environment Variables are provided / handled by OpenMP runtime library. These components of OpenMP don't require special support in LLVM IR and thus are out of scope of this document.

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

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

Next two chapters describe our design proposal for representation of, correspondingly, directives and clauses.

Directives

Directives in LLVM IR are represented as calls to “llvm.omp.directive” intrinsic with a single argument referencing LLVM IR metadata. The metadata contains an identifier of a directive.

Almost all OpenMP directives are represented with two intrinsic calls: one for entry and one for exit point of the directive’s context. It is enough to have just one intrinsic call for several directives which are supposed to be enclosed in other directives (like *omp section*, which must appear only within *omp sections* context) or specify a single instruction (like *omp flush*) or are declarative (like *omp threadprivate*). Exit point for such directives is either non-existent or can be determined by examining other intrinsic calls and thus, not required to be explicitly present.

Metadata specify only one thing: type of a directive. Currently LLVM IR does not support integer or enumeration metadata types; thus, we decided to use MDString (metadata string type) to represent the type. The list of directives and their identifiers is shown in Table 1.

Table 1. OpenMP Directives

Type of directive #pragma omp …	**MDString in metadata**
parallel	OMP_PARALLEL OMP_END_PARALLEL
[parallel] for	OMP_[PARALLEL_]LOOP OMP_END_[PARALLEL_]LOOP
[parallel] sections	OMP_[PARALLEL_]SECTIONS OMP_END_[PARALLEL_]SECTIONS
section	OMP_SECTION
single	OMP_SINGLE OMP_END_SINGLE
task	OMP_PTASK OMP_END_PTASK
taskyield	OMP_PTASKYIELD
master	OMP_MASTER OMP_END_MASTER
critical	OMP_CRITICAL OMP_END_CRITICAL

Type of directive #pragma omp …	MDString in metadata
barrier	OMP_BARRIER
taskwait	OMP_PTASKWAIT
atomic	OMP_ATOMIC OMP_END_ATOMIC
flush	OMP_FLUSH
ordered	OMP_ORDERED OMP_END_ORDERED

An example of an OpenMP directive and its representation in LLVM IR:

C/C++

```
#pragma omp parallel
```

LLVM IR

```
call void @llvm.omp.directive(metadata !0)
...
call void @llvm.omp.directive(metadata !1)

!0 = metadata !{metadata !”OMP_PARALLEL”}
!1 = metadata !{metadata !”OMP_END_PARALLEL”}
```

Clauses

All OpenMP clauses can be divided into four groups:

- Ones with predefined values (*default, ordered, nowait, untied, read, write, update, capture*)
- Ones with a list of variables (*shared, private, firstprivate, lastprivate, copyin, copyprivate,* directives *flush, threadprivate*)
- Ones with a scalar or integer expression (*if, num_threads, final, collapse*)
- Compound ones (*reduction, schedule,* directive *critical*)

Clauses in LLVM IR are represented as intrinsic calls. Each intrinsic call representing a clause has one mandatory argument and arbitrary number of optional arguments. The mandatory argument references LLVM IR metadata. The metadata contains identifier of the clause (MDString) and, for some compound clauses, additional data represented as another MDString. Optional arguments reference LLVM variables or expressions.

Clauses with Predefined Values

Clauses with predefined values are represented as calls to "llvm.omp.simple" intrinsic. For this group of clauses metadata contains an MDString with a clause's identifier.

The list of clauses and their identifiers is shown in Table 2.

Table 2. OpenMP Clauses with Predefined Values

Clause	MDString in Metadata
default(none)	OMP_DEFAULT_NONE
default(shared)	OMP_DEFAULT_SHARED
ordered	OMP_ORDERED
nowait	OMP_NOWAIT
untied	OMP_UNTIED
read	OMP_READ
write	OMP_WRITE
update	OMP_UPDATE
capture	OMP_CAPTURE

Here is an example of OpenMP clause with a predefined value and its representation in LLVM IR:

C/C++
```
#pragma omp atomic read
```

LLVM IR
```
call void @llvm.omp.simple(metadata !1)

!1 = metadata !{metadata !”OMP_READ”}
```

Clauses with a List of Variables

Clauses with a list of variables are represented as calls to “llvm.omp.list” intrinsic. For this group of clauses metadata contains an MDString with a clause’s identifier.

Additional arguments of the intrinsic reference LLVM variables associated with a clause. It is important to reference variables directly in intrinsic calls and not in metadata, in order to preserve data dependency.

Each variable of a non user-defined type is represented with a single argument (referencing the variable).

Each variable of a user-defined type is represented with four arguments: one referencing the variable itself, one referencing its default constructor, one referencing its copy constructor and one referencing its destructor. References to constructors / destructors are required to correctly create and destroy private copies of variables.

The list of clauses and their identifiers is shown in Table 3.

Table 3. OpenMP Clauses with a List of Variables

Clause	MDString in Metadata
private(*list*)	OMP_PRIVATE
firstprivate(*list*)	OMP_FIRSTPRIVATE
lastprivate(*list*)	OMP_LASTPRIVATE
shared(*list*)	OMP_SHARED
copyin(*list*)	OMP_COPYIN
Directive flush(*list*)	OMP_FLUSH
Directive threadprivate(*list*)	OMP_THREADPRIVATE

Here is an example of OpenMP clause with a list of variables and its representation in LLVM IR:

C/C++
```
#pragma omp parallel private(a,b)
```

LLVM IR
```
call void (metadata, ...)* @llvm.omp.list(metadata !1, i32* @a, i32* @b)

!1 = metadata !{metadata !"OMP_PRIVATE"}
```

Clauses with Scalar or Integer Expressions

Clauses with scalar or integer expressions are represented as calls to "llvm.omp.expr" intrinsic. For this group of clauses metadata contains an MDString with a clause's identifier.

Second argument of the intrinsic references a scalar or integer LLVM expression associated with a clause. It is important to reference expressions directly in intrinsic calls and not in metadata, in order to preserve data dependency.

The list of clauses and their identifiers is shown in Table 4.

Table 4. OpenMP Clauses with Scalar or Integer Expressions

Clause	MDString in Metadata
if(*scalar_expr*)	OMP_IF
num_threads(*integer_expr*)	OMP_NUM_THREADS
final(*scalar_expr*)	OMP_FINAL
collapse(*const_integer_expr*)	OMP_COLLAPSE

Here is an example of OpenMP clause with a scalar expression and its representation in LLVM IR:

C/C++
```
#pragma omp parallel if(a)
```

LLVM IR
```
%4 = load i32* @a, align 4
%5 = icmp ne i32 %4, 0
call void @llvm.omp.expr(metadata !1, i32 %5)
...
!1 = metadata !{metadata !"OMP_IF"}
```

Compound Clauses

Compound clauses are represented as calls to "llvm.omp.compound" intrinsic. For this group of clauses metadata contains an MDString with a clause's identifier and additional data, also represented as an MDString.

Additional arguments of an intrinsic call for *reduction* clause reference LLVM variables associated with a clause (see format of "Clauses with a List of Variables").

Second argument of an intrinsic call for *schedule (static)*, *schedule (dynamic)* and *schedule (guided)* clauses references an integer LLVM expression associated with a clause (see format of "Clauses with Scalar or Integer Expressions").

The list of compound clauses along with their representation in metadata is shown in Table 5.

Table 5. OpenMP Compound Clauses

Clause	Representation in Metadata
reduction(*operator* : *list*)	OMP_REDUCTION, <*operator*>
schedule(static *[, integer_expr]*)	OMP_SCHEDULE, STATIC
schedule(dynamic *[, integer_expr]*)	OMP_SCHEDULE, DYNAMIC
schedule(guided *[, integer_expr]*)	OMP_SCHEDULE, GUIDED
schedule(auto)	OMP_SCHEDULE, AUTO
schedule(runtime)	OMP_SCHEDULE, RUNTIME
Directive critical(*name*)	OMP_NAME, <*name*>

Here is an example of a compound OpenMP clause and its representation in LLVM IR:

C/C++

```
#pragma omp parallel reduction(+ : a, b)
```

LLVM IR

```
call void (metadata, ...)* @llvm.omp.compound(metadata !1, i32* @a, i32* @b)

!1 = metadata !{metadata !"OMP_REDUCTION", metadata !"+"}
```

An Example

Here is a more complex example, demonstrating LLVM IR representation of a directive with several clauses of different types.

C/C++

```
int a, gVar;
int main() {
  int lVar;
#pragma omp parallel default(shared), private(gVar, a), if(lVar)
  {
    ...
  }
  return (0);
}
```

LLVM IR

```
@a = global i32 0, align 4
@gVar = global i32 0, align 4
define i32 @main () nounwind uwtable ssp {
  %lVar = alloca i32, align 4
  call void @llvm.omp.directive(metadata !0)
  call void @llvm.omp.simple(metadata !1)
  call void (metadata, ...)* @llvm.omp.list(metadata !2, i32* @gVar, i32* @a)
  call void @llvm.omp.expr(metadata !3, i32* %lVar)
  ...
  call void @llvm.omp.directive(metadata !4)
  ret i32 0
}
!0 = metadata !{metadata !"OMP_PARALLEL"}
!1 = metadata !{metadata !"OMP_DEFAULT_SHARED"}
!2 = metadata !{metadata !"OMP_PRIVATE"}
!3 = metadata !{metadata !"OMP_IF"}
!4 = metadata !{metadata !"OMP_END_PARALLEL"}
```

Design Alternatives

It is possible to propose several viable alternatives to representing information on OpenMP directives and clauses while keeping general design approach intact.

Some things that we considered:

a) Represent information on types of directives and clauses as constant expressions instead of MDStrings.
b) Place MDStrings with identifiers of directives and clauses directly into intrinsic calls instead of metadata.
c) Do not represent clauses as separate intrinsics with references to metadata; instead, put a list of references to metadata for all clauses associated with a directive at the end of metadata describing the directive.

As we said, all these are viable alternatives. We decided to choose what we chose and not employ the alternatives listed above in order to keep true the following three design principles:

1) Absolute simplicity and readability (including human readability). This ruled out alternatives a) and [partially] c).
2) Uniformity of representation, for both directives and clauses. This ruled out alternatives b) and c).
3) As much extensibility and openness for future changes in both LLVM IR and OpenMP as possible. This ruled out alternatives b) and c).

We believe that the only downside of our choice is larger size of LLVM IR required to represent same number of directives and clauses.

However, we consider this as a relatively minor element of our design; one might argue in favor of these or other similar design alternatives.

Conclusion

In this document we described our design proposal for OpenMP representation in LLVM IR. We believe the IR extensions we proposed are:

1) Simple

They rely on existing LLVM IR types and language constructs; the only new things are a few new intrinsics. Thus, LLVM IR consumers lacking OpenMP support can simply ignore these two intrinsics and still generate correct, albeit sequential, code.

2) Complete

OpenMP 3.1 Specification ([OpenMP31]) is fully covered.

3) Extensible

The design approach is general enough to readily incorporate future versions of OpenMP standard.

4) Enables both early and late procedurization and aggressive optimization

This is when compared with designs implying explicit procedurization done right in front-ends.

Acknowledgements

Authors would like to acknowledge valuable contributions and advice provided by Kevin B Smith, Xinmin Tian, Stefanus Du Toit, Brian Minard, Dmitry Babokin and other colleagues from Intel and Diego Novillo from Google. This *does not* automatically imply approval of this proposal by said individuals.

References

[Lattner10] Chris Lattner, Devang Patel, “Extensible Metadata in LLVM IR”. Available at: http://blog.llvm.org/2010/04/extensible-metadata-in-llvm-ir.html

[Meyer92] Bertrand Meyer, “Applying “Design by Contract”, IEEE Computer, October 1992, pp. 40-51. Available at: http://se.ethz.ch/~meyer/publications/computer/contract.pdf

[Novillo00] Diego Novillo, “Analysis and Optimization of Explicitly Parallel Programs”, PhD Thesis, University of Alberta, Edmonton, Alberta, Canada, 2000. Available at: http://www.airs.com/dnovillo/Papers/Thesis.pdf

[OpenMP11] “OpenMP Application Program Interface”, Version 3.1, July 2011. Available at: http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[Tian05] Xinmin Tian, Milind Girkar, Aart Bik and Hideki Saito, “Practical Compiler Techniques on Efficient Multithreaded Code Generation for OpenMP Programs”, The Computer Journal, September 2005, pp.588-601. Available at: http://dl.acm.org/citation.cfm?id=1095017