# The Design, Implementation, and Usage of HLIR (High Level Intermediate Representation)

ARES Project
Los Alamos National Laboratory
Oak Ridge National Laboratory

Principal Investigator: Patrick McCormick
Kei Davis, Nick Moss

August 12, 2016

## 1 Introduction

The ARES project (Abstract Representations for the Extreme-Scale Stack) aims to create a set of interoperable tools and approaches that can be leveraged by the high performance computing community to explore and incrementally move towards more effective methods for programming next-generation architectures. The High Level Intermediate Representation (HLIR), a cornerstone of this effort, provides a common representation that compiler front-ends can conveniently target to implement parallel programming constructs such as: parallel for and reduce, communication and synchronization, tasks, and more. By leveraging a common code generation and runtime system, compiler authors can reap the benefits of a refined and optimized system while working with such parallel concepts at a higher level, leaving the exact code lowering and performance tuning to the HLIR system.

HLIR, building on LLVM IR, addresses some of the deficiencies that make it unsuitable for representing such constructs at a more abstract level, recursively, and in a way that is more amenable to optimizations, both localized

and module-wide. HLIR operates seamlessly with LLVM and provides various interfaces for the compiler writer to control the code generation process. For example, HLIR will set up an LLIR function for the body of a parallel for or reduce and provides entry points for the compiler to specify that body and will automatically take care of the details of capturing any needed data used. In this way, the HLIR system can be highly customized while automating the generation of much of the needed details to finalize and lower a parallel for or reduce, task, or communication directive including parallel launching and synchronization.

Several motivating factors have contributed to our need for a higher-level IR. Traditionally, IR's, including LLVM IR are a stream of low-level instructions – their role is to provide instructions in a form that is low-level enough to be closely mapped to the underlying hardware but high-level enough such that certain architectural differences are abstracted. Such an IR typically provides no structure to bundle/aggregate meaning or domain awareness. Typically, going from an AST to an IR, entails a loss of information such as when code-generating a loop to such an IR, the meaning of the loop's components, its body, induction variable, terminating condition, and so on, are not retained. While it is possible to reconstruct loops by analyzing the IR and looking for certain features and patterns – LLVM does in fact provide such facilities, on the other hand, for more complex constructs such as a parallel for, this can be more problematic and we lose vital information that we need to properly optimize.

One of the key advantages of our system is the ease at which parallel constructs can be created and how the system interoperates with and ties back to LLVM. In contrast with LLVM's standard metadata system, which is the prevalent way of achieving such functionality, using a metadata approach requires significant amount of "glue" code. For example, this typically involves representing structures with an id then capturing this id in the metadata. More problematic, is that LLVM metadata gets discarded as it is passed through various optimization stages/passes. HLIR data, on the other hand, is persistent throughout the compilation and we can attach arbitrary data to it via attribute fields and pointers. HLIR like LLIR, for development purposes, has a convenient textual representation. To retrieve or create an HLIR module, pass a pointer to an LLIR module. From the module, a client may then create any of the top-level HLIR constructs, e.g: parallel for/reduce, tasks, communication, etc. Designated markers allow a task, or parallel/for body to be defined, and attributes or LLIR values associated with a spe-

cific parallel construct, such as: thread index, induction variables, reduction variables, etc. can be specified or retrieved.

# 2 Background

The high level intermediate representation (HLIR) is a superset of a low-level sequential IR – LLVM IR in our case. HLIR extends LLVM IR to allow parallel and other constructs to be readily captured/represented and conveniently manipulated. Remaining language-independent, multiple compiler front-ends could target HLIR to describe common parallel features of codes such as concurrency, tasks, parallel for loops, communication directives, memory locality, data layout, code structure, and more. Our HLIR adds a flexible and expressive hierarchical high-level representation to LLIR that is capable of capturing recursive definitions and attributes that cannot be readily represented in a traditional IR. One of the ways HLIR achieves this is by embedding metadata and notional intrinsics into the traditional IR that can refer back to HLIR portions, and vice versa.

HLIR is used by both the front-ends who use it to initially conveniently capture the semantics of a parallel construct and as an intermediate representation internally in HLIR's own transformation and optimization stages until it is finally lowered into ordinary LLVM IR which can then take advantage of existing LLVM optimizations and backends.

## 2.1 LLVM

LLVM is a relatively recently developed intermediate representation and compilation toolchain designed for flexibility, optimization, type-safety, and machine independence by allowing multiple front-ends and target architectures, modularity by separating various analyses and transformations into discrete passes, and extensibility by allowing developers to readily insert their own analysis and transformation passes. LLVM has already been widely adopted for both research activities and industrial-strength compiler toolchains, and as such is the logical choice on which to build.

LLVM IR (LLIR) can be viewed as a machine-independent or universal assembly language and is used to encode the sequential semantics in our HLIR. LLVM backends for different native targets translate LLVM IR into native machine code. For example, the same LLIR can be used to produce

code for conventional CPUs or GPUs, in some cases, by performing minor modifications on the IR for each target.

LLVM IR is strongly-typed as contrasted with traditional assembly languages that operate on raw memory locations, whereas in LLVM *values* are associated with types, for example: integers and floating point values of varying bit widths, and pointers and compositions/structs thereof. The strong-typing features of LLVM makes LLIR considerable more portable and less error prone, as well as enabling a more rapid implementation of code generators.

LLVM code, at the high-level is organized into modules which are typically a translation unit or in one to one correspondence with a source file in the original program being compiled. Modules are a container of global data and functions. Functions, in turn, are made up of basic blocks – a sequence of instructions that only perform control flow at the end of the basic block, namely: conditional or unconditional branches, or return.

All in all, LLVM excels at representing serial code, but does not purport to have any direct features for the representation of parallel constructs, such as parallel for, tasks, concurrent sections, monitors, etc. On the other hand, it does provide some powerful facilities/hooks that we use to achieve some of our functionality in the HLIR such as metadata, intrinsics, address spaces, a code transformation framework – the LLVM pass manager – and the ability to add our own passes.

### 2.1.1 Metadata

LLVM metadata is a construct that allows arbitrary information to be attached to a module or to individual instructions and LLVM values (variables, functions, etc.) The metadata is used solely in the intermediate transformation of IR and in the generation of debug info, for example: mapping a source line to its instruction or a variable definition to its type. LLVM metadata, when it reaches the backend translation phase, is simply discarded.

### 2.1.2 Intrinsics

LLVM intrinsics have similar semantics of calls, but trigger a custom inlined code generation action. For example an intrinsic *call* may be expanded to a sequence of several IR instructions, avoiding the overhead of a normal call, or may be used to represent a call to a runtime library. The use of intrinsics

is our preferred method of (simulating) the addition of new instructions to the LLVM IR.

LLVM includes a powerful mechanism by which custom intrinsics can be readily defined. It is possible to add custom instructions to LLIR, however doing so entails several challenges and pre-measures that must be taken. For our purposes, many of the benefits that could be achieved by adding new instructions can be accomplished with intrinsics.

### 2.1.3 Address Spaces

LLVM provides a mechanism by which an address space can be associated with a pointer type. LLIR itself does not assign any meaning to a specific address space integral identifier – the semantical handling of this attribute is delegated to the backend in question. For example, using the NVPTX GPU backend, address space 1 is associated with global memory, whereas address space 3 is used to mark shared memory. We use this mechanism to distinguish arbitrary physical and logical memory regions.

### 2.1.4 Passes

The LLVM pass manager provides a powerful and extensible facility whereby LLVM IR can be analyzed or successively transformed for purposes of optimization or to transform it into a form suitable for final generation by a backend. LLVM provides a framework for creating module, function, and basic block passes, operating on each LLVM construct respectively. Such passes can recognize and transform metadata and intrinsics.

## 2.2 Clang

Clang, a cornerstone of the LLVM toolchain, is an optimizing C/C++ compiler that generates LLIR code. In our prior experience in working with the internals of Clang, we have found Clang AST's to be difficult to construct and modify, and by design, once they reach the code generation phase are immutable. Our HLIR is intended to fill the gaps between such a rigid AST and assembly-like code, where we need both the low-level sequential instruction level representation as well as the ability to readily represent and modify AST-like structures for our parallel constructs.

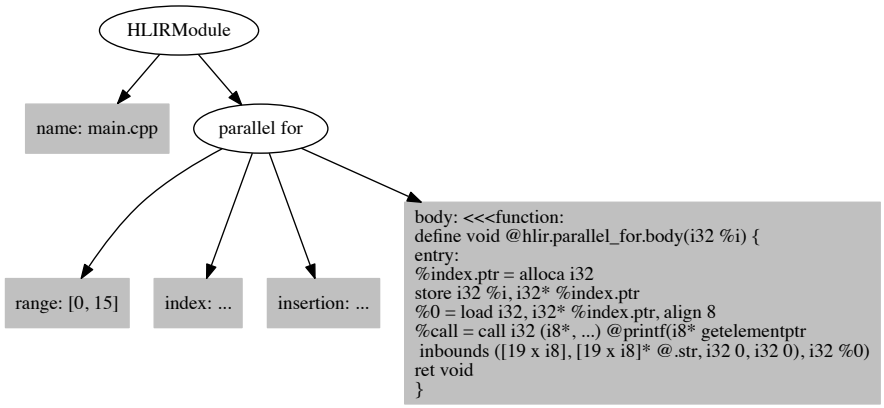
# 3 HLIR Design

An HLIR *module* corresponds to either an entire translation unit (*module*), or a sub-portion of a module. HLIR modules are in one-to-one correspondence with an LLIR module. Internally, HLIR maps an LLIR module pointer to an HLIR module so that an HLIR module can be readily retrieved given an LLIR module or created and paired with its LLIR module counterpart.

HLIR contains recursive or nested HLIR-specific constructs with LLVM IR at select leaf nodes of these constructs. HLIR modules are designed with the capability to be easily merged. For example, one document that contains definitions for a parallel HLIR construct can be merged into the scope of another HLIR module.

The HLIR is implemented as a set of interoperable C++ classes designed to provide convenient in-memory manipulation. Like LLIR, these classes can also be outputted as a human-readable textual representation. The HLIR defines a number of node types which make up the HLIR representation: leaf nodes include symbols, strings, numeric types, an LLVM IR function (which can represent an arbitrary section of code but is packaged into a proper function, encapsulating its dependencies). Recursive nodes allow multiple nesting within the representation, for example: mapping a symbol to another node (a symbol key/value pair), or a sequence of nodes, referenced by position instead of a named symbol. Both maps and sequences allow for heterogenous node value types.

The following HLIR tree representation depicts of a portion of an HLIR module containing a simple parallel for:




## 3.1 Top-level Information

The top-level HLIR section records data that pertains to the HLIR document as a whole. We capture such information as document type (module or function), original source language, HLIR version number, etc.

## 3.2 LLVM IR sections

An HLIR document may contain multiple leaf nodes that hold LLVM IR. We use LLIR's metadata facilities to tie IR fragments back to HLIR-specific constructs. Metadata that begins with `!hlir` is taken to denote an HLIR construct. So for example, a function call which has the metadata label `!hlir.task1` would reference the task1 section of the top-level, document. We can further refer to nested HLIR specifications with `!hlir.task1.dependencies` which would refer to the dependencies HLIR description of this task.

HLIR allows arbitrary fragments of LLIR to be captured - however, we place the restriction that such references to smaller fragments must be properly wrapped into an LLIR function which passes in its dependencies as function parameters – and does not reference and globals. We enforce that all LLVM nodes are valid LLIR as a unit.

We use LLIR intrinsics for constructs such as communication directives, e.g., as point-to-point or collective send/receives, similarly to MPI. The purpose of such constructs is to capture the semantics of parallel constructs,

such as communication, at a high level, irrespective of how the underlying parallel construct will be implemented or lowered to a specific runtime.

## 3.3 Memory Locality

LLIR address spaces to describe the locality of memory. Any LLVM pointer can be marked with an integral memory space identifier. We include an address space HLIR section that maintains the HLIR mappings. For example, address space = 1, might be mapped to a specific logical region in Legion whereas memory space 2 might mean a NUMA domain-level memory space.

## 3.4 Parallel Constructs

In this section, we describe at a high-level how HLIR is used to represent specific parallel constructs, using tasks, parallel for, and communication as examples. A parallel construct in general, is marked by a name corresponding to a keyed symbol under the top-level HLIR node, for example: `!hlir.task1`.

Under such a section, there exists a type specifier that indicates that such a section is a task construct, for example. Then for each parallel construct type, there are a number of required or optional attributes and an imposed structure for that particular type.

### 3.4.1 Thread Pooling and Data Capturing

Our tasking and parallel for/reduce mechanism relies upon bundling up the body of such constructs into an IR function which is code generated and queued to a common thread pool. The queued function is passed runtime arguments such as priority and synchronization objects as well as application-specific structures that capture any non-local data that the body uses. From the perspective of a client leveraging a parallel construct, capturing happens automatically, with the complexity deferred to the HLIR lowering pass. This means that when performing the code generation for a parallel construct, the code generator can ignore that such values were declared in an outer or persistent scope. To accomplish data capturing, the HLIR pass detects which IR values are referenced within the body function but which are defined externally. It creates a `struct` for those values and any references to those values are translated to offsets or `GEP` instructions into this struct.

### 3.4.2 Tasks

A task is much like an ordinary function in LLIR, but adheres to a contract in the way it uses its inputs/outputs and is restricted in how it uses global state because tasks are designed to be launched asynchronously and run in parallel. A task has an associated future in which is *forced* or waited upon when the associated value of the future is later used. An HLIR task section tracks several pieces of information associated with it: it records the future(s), and potentially later: read/write permissions of input/output parameters (perhaps as gathered or determined by an HLIR analysis pass), and so on.

HLIR allows us to create a dependency graph of the data and read/write usage of tasks in order to coordinate the asynchronous launches across multiple tasks. The representation of the dependency graph, which is not possible using existing LLIR facilities, is easily represented with our recursive HLIR constructs and can be propagated across multiple stages of the code generation process.

### 3.4.3 Parallel For

Parallel for is one example of how we use a fragment (HLIR function node) of LLVM IR to mark and encode the body of the parallel for. Parallel for loops are denoted in IR as an intrinsic (`hlir.parallel.for`) with metadata that links back to the appropriate HLIR section, where the body has been lifted into an LLVM function, and records other dependency and variable usage similar to a task. Much like a task, the parallel for HLIR section also records which variables as read/write within the loop.

One of our potential front-ends for HLIR is Kokkos, which includes a parallel for construct whereby a lambda or function or specified. In the Scout language for have a similar construct, forall for operating on entities of a computational mesh in parallel. Parallel for is a common idiom seen across many parallel languages. By targeting HLIR to encode the semantics of a parallel for, we can take advantage of common representation by which multiple front-ends target a common HLIR parallel for then within HLIR varying backends and specific runtimes can be utilized, taking advantage of a lowering process which has been thoroughly optimized to execute on a broad range of potential targets, including: GPU, threads, distributed, etc.

#### 3.4.4 Parallel Reduce

From the perspective of code generation and HLIR-wise, a parallel reduce is much like a parallel for. We generate an LLIR function corresponding to the body of the parallel reduce and perform capturing of any referenced data. The difference being that within the body we have a reduction operator and on the left-hand-side and are producing some value on the right-hand-side to either perform a sum or product. The reduce body gets passed in any captured data wrapped up into a struct and returns the value on the right-hand-side. We have implemented a reduction algorithm, which is also code-generated for performance reasons and the outermost part which gets queued to $M$ threads in the thread pool and each of these calls into the body function and computes partial sums (or products) on a slice of the index space. After computing the partial sums, the reduce algorithm combines the partial sums in an order $\lg(n)$ fashion.

#### 3.4.5 Communication

Communication directives use intrinsics and metadata linked back to HLIR sections to capture the semantics but not implementation details of communication patterns. They could ultimately, internally code generate to use varying targets such as MPI, GasNet, or Realm, for example. For this reason, they capture enough metadata about variables and their associated address spaces (the HLIR address space section) so that the lower level code generation phase can target a specific implementation or runtime layer with little configuration in the parallel compiler's targeting to HLIR.

### 3.5 Runtime Implementation

#### 3.5.1 Argobots Thread Pool

## 4 HLIR Usage and Implementation

In the preceding sections, we described the overall design of HLIR and how it may be used by a front-end to target parallel constructs. In this section, we describe some of the details of how HLIR is implemented and utilized by a front-end and how it is ultimately transformed by our backend stages to ordinary LLIR in conjunction with calls to our runtime for immediate backend code generation.

## 4.1 Usage Overview

HLIR is implemented using C++ 14, taking advantage of modern C++ features. As described, HLIR was created to interoperate exclusively with compilers utilizing LLVM for code generation. A compiler wishing to target HLIR should link with the HLIR libraries and include the appropriate HLIR headers. Then, at any point during the code generation process, the static method `HLIRModule::getModule()` can be called to either get or create the HLIR module associated with the LLIR module being code generated. Once an HLIR module has been obtained, several methods are available which can be called to create HLIR parallel constructs such as: `createParallelFor()`, `createTask()`, etc.

## 4.2 HLIR nodes

HLIR's recursive representation is provided by a hierarchy of *nodes* which are briefly described here. One should consult HLIR.h for a complete reference.

- `HLIRNode` - the base class in which all other HLIR nodes derive from. An HLIR node may be a child of at most one other node and as such it has an associated parent. Nodes are either recursive or leaf nodes. A recursive node can hold a heterogeneous collection of children nodes.
- `HLIRScalar` - the base class for simple scalar nodes such as `HLIRString`, `HLIRInteger`, etc. These are simple wrappers for scalars such as string, integer, floating point values, etc.
- `HLIRSymbol` - a lexical symbol, internally stored as a string.
- `HLIRVector` - a vector of heterogeneous nodes; used for storing positional information.
- `HLIRMap` - a recursive key/value store where keys are symbols and values are heterogeneous nodes.
- `HLIRFunction` - an LLIR Function and HLIR-specific convenience methods.
- `HLIRValue` - an LLIR Value and HLIR-specific convenience methods.

- `HLIRInstruction` - an LLIR Instruction and HLIR-specific convenience methods.
- `HLIRModule` - corresponds to an LLIR module and holds HLIR-specific parallel constructs and convenience methods and HLIR-specific metadata about the module as a whole.
- `HLIRTask` - a task which is tied to an ordinary LLIR function.
- `HLIRParallelFor` - a parallel for loop which holds an ordinary LLIR function as its body, instruction insertion point for producing code for this body, and parallel iteration variable and ranges.

## 4.3 HLIR Runtime

Our runtime is a C-based ABI interface whose implementation is in C++. The runtime consists of memory allocators, a thread pool and synchronization mechanism via *virtual semaphores*. Internally, LLIR functions are created which wrap normal calls to implement constructs such as tasks in such a way that their arguments are bundled into a closure which can then be queued to the runtime's thread pool.

## 4.4 HLIR Lowering Process

One of the major benefits of HLIR is that it provides high-level constructs and a convenient interface for interfacing with LLIR. The HLIR module pass is responsible for transforming an HLIR module and LLIR IR which ties to the HLIR module with metadata and intrinsics into ordinary LLIR with calls to our runtime. The lowering process is nearly automatic, and the creator of a front-end targeting HLIR need not be aware of the internal details, only a familiarity with the HLIR interface we provide. For example, all that is required of the front-end in targeting tasks is to specify that a certain LLIR function is a task by creating an `HLIRTask` on the `HLIRModule`, then the runtime handles the details of transforming calls to that function into task launches and awaiting futures when that future's value is required in LLIR.

## 4.5 Frontend

We have implemented a Clang-based C++ frontend with ARES-specific extensions to support parallel for/reduce, and tasks. The following code sections briefly demonstrate various usages of these constructs.

### 4.5.1 Tasks

The `task` keyword is placed at the beginning of a function declaration to mark it as a task, as this example that computes the n-the Fibonacci number in parallel demonstrates:

```
1 task int fib(int i){
2   if(i <= 1){
3     return i;
4   }
5
6   return fib(i - 1) + fib(i - 2);
7 }
```

As task is then invoked as if it were a normal function.

### 4.5.2 Parallel For

```
1 float A[SIZE];
2
3 for(auto i : Forall(0, SIZE)){
4   A[i] = i;
5 }
```

### 4.5.3 Parallel Reduce

```
1 float sum = 0.0;
2
3 for(auto i : ReduceAll(0, SIZE, sum)){
4   sum += 1.0;
5 }
```

### 4.5.4 Communication Directives

We currently do not have any frontend extensions pertaining to communication as the functionality required is easily achieved as normal functions.