Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelorarbeit

Compiler assisted translation of OpenMP to MPI using LLVM

vorgelegt von

Michael Blesel

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6443269
Erstgutachter:	Dr. Michael Kuhn
Zweitgutachter:	Prof. Dr. Thomas Ludwig
Betreuer:	Dr. Michael Kuhn, Jannek Squar

Hamburg, 2017-14-10

Abstract

OpenMP and MPI are the two most commonly used parallelization APIs in the field of scientific computing. While OpenMP makes it relatively easy to create parallel software, it only supports shared memory systems. MPI software can be executed on distributed memory system and therefore offers higher scalability options. Unfortunately MPI software is more difficult to implement.
This thesis discusses an approach to automatically translate OpenMP programs into MPI programs. With help of the tools provided by the LLVM Project, a transformation pass for the LLVM compiler is developed, which replaces OpenMP with MPI during the compilation of a program.

Hic sunt dracones -
Here be dragons

Contents

1 Introduction

High performance computing (HPC) is a field of computer science that is concerned with operating and programming for computer clusters. It is mainly used by the scientific, engineering and business communities to solve computational problems that are too complex to be tackled by a standard desktop system. A high performance computer is made up of multiple computing nodes, which are connected by a network.
To make good use of these distributed systems, the software developed for them has to be highly parallelized. The two most commonly used APIs for developing parallel programs in a HPC environment are *OpenMP* and *MPI*. *OpenMP* is a parallel programing API, which makes the compiler parallelize code sections by adding simple preprocessor directives to the source code of an application. It is often used by researchers in their software to achieve a performance gain.
In the world of high performance computing, scalability is an important factor. HPC systems might offer a high number of computation nodes, but not all parallel software automatically scales well with just increasing the number of CPUs it is run on. *OpenMP* only works on a shared memory system and therefore cannot be run on multiple nodes of clusters, which most of the time have a distributed memory architecture.
MPI, the *Message Passing Interface*, provides operations for inter-node communication and is therefore used to write parallel software for big computation clusters. A drawback of *MPI* is the higher programming effort, it requires from developers.
This thesis discusses the creation of a tool for the automatic translation of *OpenMP* programs into *MPI* programs. Using the LLVM compiler, a *Transformation Pass* for the translation of *OpenMP* to *MPI* code, at compile time, is developed.

1.1 Motivation

The motivation for a tool with the ability of translating *OpenMP* into *MPI* stems from the wish to achieve greater scalability for already existing *OpenMP* software. Many programs in the scientific computing community are build with *OpenMP*, because it gives researchers, who are not experts in parallel programming, the ability to develop parallel programs with a good performance. But these *OpenMP* programs are limited to be run on shared memory systems and therefore cannot fully use the resources of a multi-node supercomputer. To write software that can be run on multiple computing nodes, *MPI* is needed. The drawback of *MPI* is that it needs much more work and know-how to implement an *MPI* program.
The automatic translation of *OpenMP* into *MPI* would make it possible to execute original *OpenMP* software on as many nodes as needed, without having the work of

rewriting the software manually.
In this thesis, the LLVM compiler infrastructure is used to implement this. LLVM provides extendable compiler libraries for the modification of code at compile time.
A tool that compiles *OpenMP* programs into *MPI* executables would be very helpful, since it does not require any extra work on the users side. It is just a different method of building the program that automatically achieves the intended goal.

2 LLVM

2.1 The LLVM Project

Figure 2.1: LLVM Project logo

LLVM is a compiler infrastructure project written in *C++* [Pro17a]. It started out as a research project at the University of Illinois and since has become a big collection of compiler tools, which are being used in open source projects, big software companies and academic research.
The main idea behind the development of LLVM was to build a compiler that uses a *static single assignment* (SSA) compilation strategy and is programming language independent. An explanation of SSA can be found in Section 2.2. This was achieved by creating the LLVM assembly language [Pro17c], which is used as a intermediate representation (IR) for the program code inside the whole LLVM compiler. This intermediate representation can be formed out of any high-level programming language by a corresponding front-end. All optimizations which are performed during compilation are done on this intermediate representation.
The software architecture of LLVM is very modular. This has led to the development of multiple sub-projects, which are all combined under the name *The LLVM Project*.
In comparison to LLVM most other compiler projects have a more monolithic software structure, which means that the different parts of these compilers are often strongly dependent on each other. This makes it hard to modify or reuse parts of their code. For example non-LLVM optimizers might need direct communication with the compiler front-end to perform specific optimizations. LLVM on the other hand was developed with the intent of providing a set of reusable libraries for compiler developers. It is as loosely coupled as possible, so that all of its parts can be reused by developers for their compiler related projects.

Figure 2.2: The three stages of a compilation process [AB12]

Figure 2.2 shows the three major stages of a compilation. The front-end translates the input source code into the intermediate representation used inside the compiler. The optimizer analyzes and transforms the code to give it a better performance. The back-end generates the actual machine code for the target architecture. While other compilers may treat these three steps as one big process, they are much more independent from each other in the LLVM compiler. Specifically for LLVM this modular approach means, that there can be different front-ends for all kinds of programming languages and their only interaction with the optimizer is the IR code they feed to it. There are also different back-ends for different target architectures, that only take IR code from the optimizer but do not communicate with it in any other way. This modular structure of LLVM has lead to many different sub-projects, which are all combined under the name *The LLVM Project*. The following ones are used in this thesis:

- **LLVM Core:** The LLVM Core contains the LLVM optimizer and the code-generator. The optimizer is the main part of the compiler and uses *LLVM Passes* to perform transformations or analysis on IR code. It provides the code optimization strategies which are used by modern compilers to achieve good program performance. The code-generator creates machine code for the target hardware architecture and supports many different CPUs.
  This thesis uses LLVM version 4.0.1.

- **Clang:** Clang is LLVM's own *C/C++/Objective-C* compiler. It translates source code into the intermediate representation used by the LLVM optimizer. Clang's goal is to provide a modern front-end to LLVM for all C-family languages. It tries to give very clear and precise error and warning messages during compilation. Like the rest of the LLVM projects it is also build very modular and provides an API for developers to create their own analysis tools on top of it.
  This thesis uses Clang version 4.0.1.

- **OpenMP:** LLVM also provides its own implementation of the OpenMP standard to use with Clang. It supports the OpenMP 3.1 standard since the release of Clang 3.8.0. Code compiled by Clang with the `-fopenmp` flag is linked against the LLVM OpenMP Runtime Library [Pro17d].

- **lld:** lld is the linker used by LLVM/Clang. It was developed to be a replacement for the standard GNU system linkers and promises faster linking times, especially on multi-core architectures.

LLVM is used by many big corporations or scientific researchers. For example, all operating systems produced by *Apple* are build with LLVM. Clang is the default compiler

on *Mac OS* and *Apple*'s own development environment *Xcode* uses and is build with LLVM.
A reason for LLVM being used in academic research - and in this thesis - is its focus on providing an API for developers to customize it. It is relatively easy to create custom *LLVM Passes* and thereby analyze or modify a program during its compilation. The intermediate representation used inside the compiler can also be printed in a human readable form. This makes it possible for the developer to keep track of the different transformations of the code inside the optimizer and enables them to write their own. The next sections of this chapter will take a closer look at the parts of the *LLVM API* and the *LLVM assembly language*, that will be used in the later chapters.

2.2 The LLVM intermediate representation

The LLVM intermediate representation, commonly referred to as IR, is a low level, assembly-like language, that is used in the whole LLVM optimizer. It is supposed to give compiler developers the possibility to implement code optimizations on a language that is detached from the original programming language and not yet concerned with the target hardware architecture. There are three interchangeable forms of LLVM IR. First, as a in-memory construct, that is passed through the compiler. Second, as a byte-code format that can be written to disk in a compact form, which can be loaded back into the compiler. Third, as a human readable form that gives a developer insight into the transformations done to the code by the compiler.
LLVM IR is a strongly typed, static single assignment (SSA) based programming language. Single static assignment form is an attribute of many intermediate representations. In SSA every variable is defined exactly once in the code and can only be used after its definition. A change in a variable's value produces a new version of the variable, which is most of the time represented by giving the variable names indices. Listing 2.1 illustrates the difference between normal *C/C++* code and its SSA form. We can see that each assignment to `x` creates a new version of the variable. To handle branching, so called phi-nodes are added to the language. The ϕ `(x_2, x_3)` instruction in line 9 means that, if the code reached the instruction from the first branch `y_1` gets `x_2` assigned as its value. If the instruction was reached from the second branch, `y_1` gets assigned `x_3`.

```
1 Non-SSA form      | SSA form
2 ------------------------------------
3   x = 0;          |  x_1 = 0;
4   if(x == 0)      |  if (x_1 == 0)
5     x = x + 42;   |    x_2 = x_1 + 42;
6   else            |  else
7     x = 0;        |    x_3 = 0;
8                   |
9   y = x;          |  y_1 = phi(x_2,x_3);
```

Listing 2.1: Example of code in SSA form.

The SSA form simplifies data flow analysis and many optimizations, which is why it is used by modern compilers.

```
1  int main()
2  {
3    int a = 42;
4    print_function(a);
5    return 0;
6  }
7  ------------------------------------------
8  ; Function Attrs: noinline norecurse uwtable
9  define i32 @main() #2 {
10   entry:
11     %retval = alloca i32, align 4
12     %a = alloca i32, align 4
13     store i32 0, i32* %retval, align 4
14     store i32 42, i32* %a, align 4
15     %0 = load i32, i32* %a, align 4
16     call void @_Z14print_functioni(i32 %0)
17     ret i32 0
18 }
```

Listing 2.2: Example of LLVM IR code for a simple program

Listing 2.2 shows the IR code generated for a simple *C++* program's main method, in which an integer variable *a* is created, assigned a value and is then passed to another function. Most features of LLVM IR, that need to be explained for later chapters of this thesis, can be shown in this code snippet.
LLVM IR is strongly typed, which means that every instruction has to explicitly state the type of its operands. Following are the most important types that the LLVM IR type system provides:

- **Integer types**: Integer types in LLVM IR are represented in the form of `iN`, where `N` stands for the length of the integer in bits. A normal *C/C++* **`int`** translates into `i32`.

- **Floating point types**: Floating point numbers are represented in the form of keywords like `half` (16-bit), **`float`** (32-bit) and **`double`** (64-bit).

- **Pointer types**: Pointers are represented in a *C*-like form with the use of the `*` character. An `i32*` type is a pointer to an integer. An `i32**` type is a pointer to a pointer of an integer. LLVM IR does not differentiate between *C/C++* pointers and references. Both are represented as pointers.

- **Array types**: Arrays are represented in the form of `[N x type]`, where `N` is the size of the array and `type` is the array's type.

- **Vector types**: Vector types, that can be used for single instruction multiple data (SIMD) operations, are represented in the form of `<N x type>`.

- **Structure types**: Structure types are represented in the form of `type {type-list}`, where `type-list` is a list of the struct member's types. A `type {i32, i32*}` type would refer to a struct with an integer as its first element and an integer pointer as the second.

- **label types**: Label types are represented with the `label` keyword. Labels are used for code branching. When the control flow of the program has to jump somewhere in the code it is accomplished with labels.

The overall structure of IR code is that all code is inside a *Module*. A *Module* represents a translation unit, which in *C/C++* equals a source file with all its `include`s. Inside the *Module* are all global variables and functions. Functions are constructed from *Blocks*, that are labeled. *Block*s contain the instructions the program is made of.
In lines 8 and 9 of Listing 2.2 we can see the definition of the program's main function. Line 8 shows the function's attributes. The `;` character is used for comments in LLVM IR. We can see that the function's return type is integer and it does not expect any arguments.
The `@` character in front of the function name tells us that the function name is a global identifier. LLVM IR differentiates between global and local identifiers. Global variables and functions are global identifiers and have the `@` character at the beginning of their name. Register names and types are local identifiers and have the `%` character at the beginning of their names. As we can see in lines 11, 12 and 15 of Listing 2.2 all created values have the `%` character added to their names.
Local values in LLVM IR are always stored in registers. LLVM IR acts on the premise that there are infinite registers available. The actual mapping to specific CPU registers is done later by the back-end that translates the IR into machine code for the specific target architectures. Unnamed values that are automatically created by the compiler, like `%0` in line 15 of the example get assigned numbers as names.
In line 10 of the example we see the `entry` label, which makes the first block of the function. The first block of a function is always labeled as `entry`. Additional blocks would be created for constructs like an `if` statement or a loop.
In line 11 we see the allocation of an integer value by using the `alloca` instruction. The allocated value gets assigned the name `retval`.
The following list will highlight some important LLVM IR instructions, which will be used in Chapter 4.

- **alloca**: The `alloca` instruction allocates memory for the given type on the stack. The instruction takes the type and number of elements as arguments. If no number of elements is specified it defaults to one. The returned value of an `alloca` instruction is a pointer to the allocated value. `%a = alloca i32` creates a integer on the stack-frame and returns a pointer to it, which is saved in `%a`.

- **store**: The `store` instruction writes a value to a given address. As first argument it takes the value that is to be stored and as second arguments the target address. `store i32 42, i32* %a` stores the integer value 42 at the address of `%a`.

- **load**: The `load` instruction is a read from memory. It takes the type of the loaded value and a pointer to it as arguments and returns the loaded value. `%value = load i32, i32* %a` loads the value that is stored at the address to which `%a` points and assigns it to `%value`.

- **call**: The `call` instruction calls a function and returns the return value of the function. `%call = call i32 @test_func(i32 %a)` calls a function with the signature `i32 test_func(i32)` and assigns the functions return value to `%call`.

- **getelementptr**: The `getelementptr` instruction computes an address pointer for accessing an object. It is used to access elements of an array or a struct. To compute the right address to a *C++* statement like `arr[1]`, the IR code looks like this: `%arrayidx = getelementptr inbounds [4 x i32], [4 x i32]* %arr, i64 0, i64 1`. The first argument is the type of the accessed array or struct. The second argument is a pointer to the base address of the object. The last two arguments are used to get the element at the right index. In this example the result of the instruction is a pointer to the second element of the array.

2.3 The LLVM API

If a developer wants to add customization to LLVM, their first point of reference should be the *LLVM doxygen pages* on the official website of the LLVM Project [Pro17b]. These pages compose the automatically generated documentation of the LLVM source base. There are also many different guides and tutorials [Pro17e] helping new developers getting acclimated with the LLVM infrastructure on the LLVM website.
The following sections will highlight the most important parts of the LLVM API that will be used in Chapter 4.

2.3.1 LLVM Passes

The best method for adding customized functionality to the LLVM compiler is creating a new *LLVM Pass*. Every transformation or analysis inside the LLVM optimizer is done by so called *LLVM Passes*. All passes that can be called by the optimizer exist in the form of library files. Most Passes are completely independent from each other and perform one specific task on the code.
Figure 2.3 shows the steps, which the code of a program goes through during a compilation with LLVM. After being translated into LLVM IR code by Clang, the code gets passed to the optimizer. The optimizer schedules the order of all passes that have to be executed and then pipes the IR code from pass to pass until all optimizations are done and the code is given to the back-end. A pass receives the current state of the compiling code in

the LLVM IR format as input. It then modifies or analyzes the code and returns the new IR as output to the optimizer.

A good example for an LLVM Pass is the **`-inline`** pass, which is part of the standard

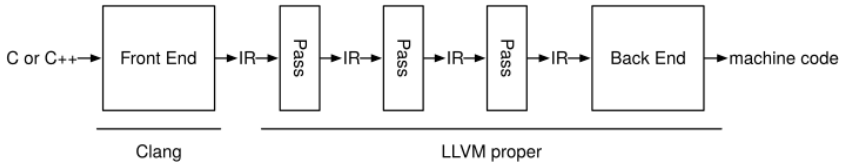

Figure 2.3: Transformations of code during a llvm compilation process [Sam15]

LLVM optimizer. It is used to inline function calls to achieve faster run-times of programs. All optimizations that a user can activate in a modern compiler, with for example the `-O3` flag, are implemented as passes in LLVM. The optimizer executes them one after the other and gives the final IR to the code generator to translate it into machine code.
LLVM divides passes into three categories:

1. **Transformation Passes:** Transformation Passes modify the code. Optimizations like the formerly mentioned **`-inline`** are Transformation Passes.

2. **Analysis Passes:** Analysis Passes collect information about the program. They can be used to give information about the code's structure to other passes or to print out analysis data, which can be used by other programs.

3. **Utility Passes:** Passes that do not fall into the already mentioned categories are called Utility Passes. One example is the `-verify` pass, that verifies IR code and can be used for testing while developing a Transformation Pass.

The basis of all passes is the *Pass* class. All passes have to be derived from this base class. LLVM provides several subclasses of the *Pass* class, that custom passes can inherit from. These subclasses specify the way the pass will be used. This information is used by LLVM to schedule the order in which it will execute them in the optimizer.
All passes in the optimizer are managed by a *PassManager*. The *PassManager* has a list of all existing passes and is responsible for scheduling, and executing them. To use a custom pass, it has to be registered in the *PassManager*. During registration the user can give instructions on where to add the pass into the order of passes.
Each pass class provides different virtual methods, which can be overridden by the developer while creating their own pass. The two pass classes relevant to this thesis are:

1. A ***FunctionPass*** is executed for every function in the program. It is used to modify the contents of a function but is not allowed to modify anything outside of it. It provides the virtual function **`bool runOnFunction(Function &F)`**, that is called automatically by the optimizer for every function in the program. Its return value is a **`bool`**, that shows whether the function has been modified.

2. The ***ModulePass*** is executed for every translation unit. A translation unit in *C/C++* equals a source file with all its included files. A Module pass can practically modify the whole program. It can modify or remove functions and add new functions. The developer can override the virtual `bool runOnModule(Module &M)` method, which gets called for every translation unit. A *ModulePass* also has to return a bool indicating whether the contents of the module have been modified.

```
1  using namespace llvm;
2  namespace {
3    struct HelloPass : public FunctionPass {
4      static char ID;
5      HelloPass() : FunctionPass(ID) {}
6      virtual bool runOnFunction(Function &F) {
7          errs() << "Pass executed on function: " <<
               ↪ F.getName() << "\n";
8          return false;
9      }
10   };
11 }
12 char HelloPass::ID = 0;
13 static void registerHelloPass(const PassManagerBuilder &,
14                               legacy::PassManagerBase &PM) {
15   PM.add(new HelloPass());
16 }
17 static RegisterStandardPasses
18   RegisterMyPass(PassManagerBuilder::EP_EarlyAsPossible,
19                  registerHelloPass);
```

Listing 2.3: Example of a custom FunctionPass implementation

Listing 2.3 shows the implementation of a very basic *FunctionPass*. It is executed for every function in the program and prints out the name of the function it is currently working on. The basic structure of this pass is after the examples given in [Sam15]. The needed pre-processor `#include` statements are omitted for space reasons.
To run the pass on an example program it first has to be build into a shared library. After the pass is build it can be added to the compilation process using *Clang*. To achieve this the developer has to add the `-Xclang -load -Xclang <path to pass.so>` flags to the normal *Clang* call. This adds the pass to the compilation process and it gets executed by the compiler. The output from using this pass during the compilation of a simple "*Hello World*" program is `Pass executed on function: main`.
This example pass does not modify the program. The output produced by the pass happens only during compilation and does not affect the created binary in any way. It could be categorized as an *Utility Pass*.

2.3.2 Additional LLVM classes

This section will focus on the structure of the *LLVM API* and its most important classes. These need to be understood to follow the implementation of the pass developed in Chapter 4.
The LLVM source base makes heavy use of inheritance. There are a few base classes, providing generic methods for nearly all kinds of objects a developer will encounter when working on modifying IR.

The Module class

The *Module* class represents a translation unit of IR code. It holds all top-level information about the program like global variables, functions and a `SymbolTable` containing the names of all objects. It provides iterators and iterator ranges to the developer to access all functions or global variables in the module, which can be used to loop through them in a pass.

The Function class

The *Function* class represents a function in the IR code. It is structured similarly to the *Module* class in the way that it contains a list of all *BasicBlock* objects, which make the body of a function. It also provides a list of the function's arguments. Like all container classes in LLVM it provides iterators and iterator ranges to access its elements. Listing 2.4 shows how to loop through all *BasicBlock*s of a function using a *C++11* range-based for loop.

```
1 Function F = ...;
2 for(auto &basicblock : F)
3 {
4     basicblock.dump();
5 }
```

Listing 2.4: Example for looping through all BasicBlocks of a Function F

The Type class

Since LLVM IR is strongly typed, every object has a specific type. The *Type* class represents the type of an object. Together with the *Value* class it is one of the most fundamental parts of the LLVM source base. All specific types that an object in LLVM IR can have, are derived from this base class. It mainly provides boolean methods, like `isIntegerTy()`, for the developer to check whether an object is of a specific type.

The Value class

The *Value* class is the base class for any kind of typed object encountered in LLVM IR. It is the base class to a big part of the LLVM source base, ranging from *Constants* and

Instructions to *Functions*. Every *Value* has a *Type* attribute that can be accesses by the `getType()` method. One of its most useful features is a list of *Users*. Every operation that refers to the value is an *User* of this value. This allows the developer to track all places in the code where a specific value gets used without having to search for them manually.

The IRBuilder class

Using the *IRBuilder* class is the main method of generating IR code. An *IRBuilder* object accepts a pointer to an *Instruction* in its constructor call. It then creates an *InsertPoint* right before the instruction where it can insert new instructions. The insert point stays the same until it gets changed manually, which means that inserting several instructions in a row achieves the intuitive result of inserting them all in front of the original instruction given as insert point. The *IRBuilder* interface provides methods for creating any kind of instruction, like `CreateCall` for inserting a function call, `CreateAlloca` for allocating a value or `CreateMul` for a multiplication operation.

Utility functions

- The `dump` function prints out the IR of an LLVM object during the execution of the pass. It is a member function of nearly every LLVM class and is mainly used for testing and debugging purposes. It makes it much easier to follow the inner workings of a pass.

- The `errs` function is an LLVM native replacement for the *C++* `std::cerr`. It is a stream to the *standard error* and should also be used for testing and debugging purposes.

- The `dyn_cast` function is a dynamic cast for LLVM objects. It is often used to cast a LLVM `Value` object into an object of a specific sub-class of `Value`. Many LLVM functions only work with objects of the `Value` class, since it is the most general class in LLVM. This allows it to write general functions for different types of objects.
  When an operation needs to know the specific sub-class of a `Value` object, a `dyn_cast` can be used to cast it down the inheritance tree.

3 OpenMP and MPI

Parallelization is one of the most important techniques in modern software development to gain better performance for programs. This stems directly from the kind of new hardware that is developed today. Since the performance of single core CPUs could not be increased much more from one Processor generation to the next most of the market shifted to the development of multi core processors, that have more than one processing unit on one chip [Sut].
To use this kind of hardware efficiently, a programmer has to use parallel programming techniques. The workload of programs has to be split into multiple threads or processes to make full use of all processing units that the underlying hardware provides. To make this process easier for the programmer, different tools and software libraries have been developed to support the implementation of parallelized software. The two parallelization libraries that are used in this thesis are *OpenMP* [Ope17] and *MPI* [MPI17].
OpenMP is an API for parallel programming, that uses preprocessor directives and some library functions to let code sections be parallelized by the compiler. It creates multiple threads for each parallel section and only works with shared memory. Shared memory describes the fact that the whole program can access the same memory regions. This is for example the case on a normal desktop machine, where the CPU is directly connected to the physical memory.
A major advantage of using *OpenMP* is that it is relatively easy to use. Already existing serial source code can be partly parallelized by adding *OpenMP* preprocessor directives. Most of the actual work for the parallelization is then done by the compiler, which automatically generates code for the creation of *OpenMP* threads and the parallel execution of the program. This ease of use is a reason why *OpenMP* is often used by researchers and other people, who are not experts in writing parallel software, but need to solve problems that require a large amount of computing power.
While *OpenMP* provides a great way of enhancing the performance of programs on shared memory systems, it eventually runs into problems from a scalability stand point. If the problem that the program is solving becomes big enough, that only a supercomputer can reasonably solve it, *OpenMP* often reaches it limits.
Most supercomputers have a distributed memory architecture. They consist of many nodes, which are networked together to form the supercomputer. In this case, if a program is run on multiple nodes, not all processing elements can access the same physical memory. Each node has its own memory, which is connected to the node's CPUs.
The *MPI* API makes it possible to write programs with inter node communication. A program parallelized with *MPI* consists of multiple processes of the same program that are run simultaneously. They communicate with each other by passing messages through

the network. Compared to *OpenMP* the programming effort needed to write an *MPI* program is much higher. Since the different processes do not have access to a shared memory, all changes to data, that is supposed to be shared, have to be synchronized manually by passing messages between the processes. This needs good planning from the programmer to prevent data races and deadlocks in the program. The upside of using *MPI* is that it provides great scalability. If programmed right, the program can be executed on as many nodes as the supercomputer can provide.
The next sections will give a brief overview of the features provided by *OpenMP* and *MPI*.

3.1 OpenMP

```
1  #include<omp.h>
2  #include<stdio.h>
3
4  int main(){
5      int counter = 0;
6  #pragma omp parallel shared(counter)
7      {
8          printf("Hello from OpenMP thread: %d\n",
               ↪ omp_get_thread_num());
9          counter++;
10     }
11     printf("Counter: %d\n", counter);
12 }
```

Listing 3.1: An example of a simple OpenMP program.

To use the *OpenMP* API in a *C/C++* program the `omp.h` header has to be included in the source file. Listing 3.1 shows a simple *OpenMP* program. The program prints out a message from every *OpenMP* thread that is created and makes every thread increase a counter variable by one.
In line 6 we can see the preprocessor directive that tells the compiler to parallelize the following block of code. An `omp parallel` pragma creates threads, which all execute the following block of code in parallel.*OpenMP* also provides a special pragma `omp` ↪ `parallel for`, which automatically parallelizes a *for loop* by dividing the loop's iterations between the created threads.
At the end of line 6 we see the `shared` keyword. This keyword tells the compiler that the following variables are so called *shared variables* for the parallel pragma. This means that all threads read and write to the same memory location when they access the variable. Hence the variables value is shared between all threads.
The `private` keyword is the opposite of this and indicates that the mentioned variables are *private variables*, which are thread local and cannot be accessed from another thread. An example for the use of the *private* keyword are the counter variables in *for loops*. In

a parallelized *for loop* each thread is responsible for a different range of the loop and so the counter variables have to be independent for each thread.
Line 8 shows the use of one of *OpenMP*'s special functions. `omp_get_thread_num` returns the rank of the *OpenMP* thread that it is executed in. All threads get assigned a number by *OpenMP*, ranging from zero to the number of created threads minus one.
How many threads are created by a *OpenMP* program is decided when the program is executed. The user can choose this number by setting the environment variable `OMP_NUM_THREADS`, which sets the maximum number of threads, before the program is started. Inside the code the number of created threads can be accessed by calling the `omp_get_num_threads` function.
To compile a program with *OpenMP* functionality activated one has to give the correct *OpenMP* flag to the compiler. For Clang and gcc this is the `-fopenmp` flag. If this flag is given to the compiler, it translates all pragmas into calls to the *OpenMP Runtime Library* [Pro17d] and creates a binary that can be executed normally.

3.2 MPI

```
1  #include<stdio.h>
2  #include<mpi.h>
3  int main()
4  {
5    int rank, size;
6    MPI_Init(NULL, NULL);
7    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8    MPI_Comm_size(MPI_COMM_WORLD, &size);
9    printf("Hello from MPI rank: %d\n", rank);
10   MPI_Barrier(MPI_COMM_WORLD);
11   if(rank == 0)
12     printf("Number of MPI processes: %d\n", size);
13
14   MPI_Finalize();
15 }
```

Listing 3.2: A simple MPI example program.

The *MPI* API can be used in a *C/C++* program by including the `mpi.h` header file. In Listing 3.2 we can see the source code of an example *MPI* program. The program prints out a "*Hello*" from every process and then prints the number of *MPI* processes once.
In line 5 two integer variables are declared. They are there to hold the information about which rank the current process has and how many processes are running. Like *OpenMP* numbers its threads, *MPI* also gives numbers ranging from zero to the number of running processes minus one.
In line 6 the `MPI_Init` function is called. As the function name says it initializes *MPI*. `MPI_Init` should be called at a very early stage of the program. In lines 7 and 8 we

see the two functions `MPI_Comm_rank` and `MPI_Comm_size` used. They write the rank of the current process and the number of running processes to the memory address that is passed to them as their second arguments. The first argument of both functions is an `MPI Communicator`.
Some *MPI* functions take communicators as arguments to know which processes are taking part in the operation that is triggered by the function. Communicators have a list of processes, that are part of the specific communicator. A process can be part of multiple communicators at the same time. `MPI_COMM_WORLD` is the default communicator and it consists of all *MPI* processes that are running.
In line 9 every process prints out a message including its rank. Line 10 is a call to `MPI_Barrier`. This is one of *MPI*'s functions to keep processes synchronous. When an *MPI* program is started the requested number of processes is created. Each process then starts to execute the program independently from each other. This means that there is no guarantee that the processes are are at the same parts of the code at the same time. The speed at which each process is executed depends on different external factors, like the scheduling of the operating system for each process. To make sure that all processes of an *MPI* program get to the same part of the code at the same time *MPI* provides blocking operations. If a process reaches a blocking *MPI* operation the execution of the program stops until all other processes that are taking part in the operation also reach it. `MPI_Barrier` makes processes wait until all others are also at the barrier call.
Lines 11 and 12 show an example of how we can let different *MPI* processes execute different parts of code. Since in an *MPI* program all processes run the same binary we have to declare it in the source code when we only want specific processes to execute specific instructions. This is mostly done by using an `if` statement to specify the ranks of the processes we want to execute code. In the case of our example only the process with rank 0 prints out the message in line 12.
In line 14 `MPI_Finalize` is called. It is the pendant to `MPI_Init` and has to be called to stop *MPI* and do cleanup. `MPI_Finalize` should always be called before the program finishes.

3.2.1 MPI Communication

Listing 3.2 showed a simple *MPI* program that executed print instructions in parallel. It did however not address the most important feature of *MPI*. The processes running this example program have no communication between them. As the name "*Message Passing Interface*" says, the *MPI* API provides different operations to send and receive messages between processes. *MPI* communication methods can be categorized into two approaches.

1. **Two sided Communication**: In two sided communication all processes that are part of a communication operation have to actively participate in it. One side has to initiate a sending operation, which sends a buffer of data. The other side has to call a receive operation to indicate that it will receive a data buffer. Two sided communication operations can be further divided into blocking and non blocking

operations. In a blocking communication the sending and the receiving processes both do not execute any further code until the communication has completed. In non blocking communication the processes can continue their work after the *MPI* operation call, but it is still necessary for both sides to declare that there will be a communication operation.

2. **One sided Communication**: In one sided communication, which is also called *MPI Remote Memory Access* (MPI-RMA), a process can write or read into or from the memory of another process without the target process's active participation. This is accomplished by first creating a *Window* of memory between all processes, which specifies the memory regions that can be targeted by *RMA* operations. After the window is created all processes can access the memory regions of the other processes.

The following sections will give a brief overview and explanation of the two *MPI* communication methods without going into too much detail, since this is not the main focus of this thesis.

MPI two sided communication

```
1  int counter = 0;
2  if(rank == 0) {
3    counter++;
4    MPI_Send(&counter, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
5  }
6  else if(rank != size-1){
7    MPI_Recv(&counter, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD,
       ↪ MPI_STATUS_IGNORE);
8    counter++;
9    MPI_Send(&counter, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
10 }
11 else {
12   MPI_Recv(&counter, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD,
       ↪ MPI_STATUS_IGNORE);
13   counter++;
14   printf("Counter value: %d\n", counter);
15 }
```

Listing 3.3: MPI two sided communication example program

Listing 3.3 shows an example program with *MPI* two-sided communication operations. The program increases a counter variable (line 1) by one for each process and prints out its final value in line 14.
The example introduces two *MPI* communication operations.

1. `MPI_Send` is a blocking send operation. It sends a buffer to another process. The first argument of `MPI_Send` is a pointer to the buffer that is going to be send. The second and third arguments give the number of elements that are send and their type. In our example one integer is send. The fourth argument is the destination of the message. It expects the number of the rank that is supposed to receive the message. The last two arguments are a message tag, which is optional and the communicator in the operation takes place.
   `MPI_Send` is a blocking operation. This means that the execution of the program stops until the message is received by the target process.

2. `MPI_Recv` is the counterpart to `MPI_Send`. It receives a message and stores the contents into a buffer. The first arguments is the buffer in which the received message is stored. the second and third argument are again the number and type of received values. The fourth argument is the rank of the process from which the message is received. Arguments five and six are the same as in `MPI_Send`. The last argument is an `MPI_Status` object, which can deliver information about the communication. If this is not needed it can be ignored by passing `MPI_STATUS_IGNORE`.

The example program increases the counter variable in order of processes. Access to the variable does not happen simultaneously. The increase operation is first executed by the rank 0 process. The new value of `counter` is then send to the next process(line 4). The next process has been waiting at its blocking `MPI_Recv` operation. After receiving the updated value of `counter`, the process increases it again and then sends the new value to the next process(lines 8-9). This procedure is repeated until all processes have received a value for counter and increased it again.
This procedure is necessary to achieve the correct result, because all processes have their own version of the `counter` variable. When a process modifies the value of a variable, that is needed in other processes, the new value has to be send as a message.

MPI one sided communication

```
1  int arr[size];
2  MPI_Win win;
3  MPI_Win_create(arr, size, sizeof(int), MPI_INFO_NULL,
     ↪ MPI_COMM_WORLD, &win);
4  arr[rank] = rank;
5  if(rank != 0) {
6    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 0, 0, win);
7    MPI_Put(&arr[rank], 1, MPI_INT, 0, rank, 1, MPI_INT, win);
8    MPI_Win_unlock(0, win);
9  }
10 MPI_Barrier(MPI_COMM_WORLD);
11 if(rank == 0) {
12   for(int i = 0; i < size; i++) {
```

```
13      printf("arr[%d] : %d\n", i, arr[i]);
14    }
15  }
16  MPI_Win_free(&win);
```

Listing 3.4: MPI one sided communication example program

The example program in Listing 3.4 uses one-sided *MPI* communication to modify an array. Each process writes into the array at the index of its rank.
In line 3 an *MPI* window is created. This operation is needed to set up *MPI* one-sided communication for the array.
The `MPI_Win_create` function is a collective operation. This means that all processes that want to participate in RMA operations on the window have to call this function. The function takes an address pointer to the beginning of the allocated memory for the array as first argument. The second and third arguments are the number of elements in the array and the size of each element in bytes. The fourth and fifth element are an `MPI_INFO` object and the chosen `MPI_Communicator` for the operation. The last arguments is a pointer to an `MPI_Win` object, which is used as a return parameter for the created window. After the window has been declared it can be used to perform one-sided *MPI* communication operations on it.
In lines 5-8 all processes with a rank that is not 0, write an element into the memory of process 0. For the writing operation `MPI_Put` is used. This function specifies in its first three arguments, what data is to be send. The first arguments is a pointer to the memory address of the data that is to be send. The second and third argument specify the number of send elements and their type. The fourth arguments is the rank of the target process for the operation. The fifth element specifies the displacement at which the data is supposed to be written into the window. This can be seen as the targeted index of the window. The next two arguments are, again, the number of elements that are to be written and their type. The last argument is the `MPI_Win` that the operation is executed on.
The put operation is surrounded by an `MPI_Win_lock` call, which locks access to the window in rank 0 while the put operation takes place.
In lines 11-15 the rank 0 process prints out the contents of the array. In an example with four *MPI* processes, the values of the array elements in rank 0 would be: `[0,1,2,3]`.

4 Developing an LLVM Pass to translate OpenMP to MPI

This chapter will discuss the development of an LLVM pass for translating *OpenMP* to *MPI*. We will first look at the IR code that gets created for *OpenMP* programs and analyze it. Then the basic structure of the pass is discussed. Finally, the actual implementation of the pass is explained.

4.1 Analyzing IR of OpenMP programs

```
1 int main()
2 {
3 #pragma omp parallel
4     {
5         int a = 42;
6     }
7 }
```

Listing 4.1: Simple OpenMP test program

```
1  %ident_t = type { i32, i32, i32, i32, i8* }
2  @.str = private unnamed_addr constant [23 x i8]
     ↪ c";unknown;unknown;0;0;;\00", align 1
3  @0 = private unnamed_addr constant %ident_t { i32 0, i32 2,
     ↪ i32 0, i32 0, i8* getelementptr inbounds ([23 x i8],
     ↪ [23 x i8]* @.str, i32 0, i32 0) }, align 8
4
5  define i32 @main() #0 {
6    entry:
7      call void (%ident_t*, i32, void (i32*, i32*, ...)*,
         ↪ ...) @__kmpc_fork_call(%ident_t* @0, i32 0, void
         ↪ (i32*, i32*, ...)* bitcast (void (i32*, i32*)*
         ↪ @.omp_outlined. to void (i32*, i32*, ...)*))
8      ret i32 0
9  }
10
11 define internal void @.omp_outlined.(i32* noalias
     ↪ %.global_tid., i32* noalias %.bound_tid.) #1 {
```

```
12   entry:
13     %.global_tid..addr = alloca i32*, align 8
14     %.bound_tid..addr = alloca i32*, align 8
15     %a = alloca i32, align 4
16     store i32* %.global_tid., i32** %.global_tid..addr,
         ↪ align 8
17     store i32* %.bound_tid., i32** %.bound_tid..addr, align
         ↪ 8
18     store i32 42, i32* %a, align 4
19     ret void
20 }
```

Listing 4.2: LLVM IR code of a simple OpenMP program

Before implementing the pass we first have to understand how an OpenMP program looks like after it has been translated into IR by Clang. To get the IR code of an OpenMP program we can compile it normally with Clang and add the `-emit-llvm -S -o -` flags to the compiler call.
After compiling the program from Listing 4.1 with these flags and OpenMP enabled we get the IR code that was produced by Clang and given to the optimizer. Listing 4.2 shows a shortened version of the IR code for our test program. The listing only contains the most important parts of the produced IR for brevity reasons.
As we can see in line 7 of the example, the body of the program's main function has been replaced by a function call.

```
1 void kmpc_fork_call( ident_t* loc, kmp_Int32 argc,
    ↪ kmpc_micro microtask, ... )
```

Listing 4.3: The kmpc_fork_call function

This is a call to the LLVM OpenMP Runtime Library [Pro17d]. In the LLVM compiler all OpenMP operations get replaced by calls to this library. The `kmpc_fork_call` (Listing 4.3) function is responsible for initializing the execution of a OpenMP parallel pragma. It takes the following arguments:

- `loc`: A Pointer to a struct of the type `ident_t`, which is a type from the OpenMP Runtime Library that holds source location information. The struct contains information about the identity of the current thread.
- `argc`: The number of shared variables used in the OpenMP parallel section.
- `microtask`: A Pointer to an `ompoutlined` function. These functions are created by the compiler and contain the code that is parallelized by the OpenMP pragma.
- `...`: Pointers to all shared variables that are used in the parallel section.

In our example `kmpc_fork_call` gets called with a pointer to the global variable `ident_t`, an integer with the value of zero and a pointer to the microtask function `@.ompoutlined..`

The microtask function contains the instructions that are executed by all OpenMP threads. The compiler creates a specific microtask function for every OpenMP parallel pragma in the source code. They always take the following arguments:

- `global\_tid`: global thread identity of thread executing the function.
- `bound\_tid`: local thread identity of the thread executing the function.
- `...`: Pointers to the shared variables of this OpenMP parallel pragma.

In this example the microtask function does not take any pointers to shared variables as arguments, because there are no shared variable in the example program. Inspecting the body of the microtask function we can see that the first instructions are the allocation of pointers to the addresses of the functions arguments and local variables. Next the received arguments are stored in the corresponding address variables. This pattern is true for all microtask functions. For every shared variable in a microtask function the same two instructions will be created. After all arguments have been stored in these address variables the rest of the microtask consists of the body of the original OpenMP pragma. In this case the body is only a store instruction to the local variable `%a`.

After analyzing the general structure of IR code that is created for an OpenMP program we can create a list of steps that have to be done to translate an OpenMP program into an MPI program. This thesis works under the assumption that the OpenMP program, which will be translated into an MPI program is purely parallelized with OpenMP. Hybrid OpenMP-MPI programs, that already use both forms of parallelization are not directly addressed. The used methods for translating OpenMP to MPI can of course still be applied on Hybrid programs, but the structure of the developed LLVM pass needs to be modified to take already existing MPI operations into account. The basic steps that have to be implemented in the translation pass are:

1. **Adding MPI initialization and finalization to the program**: To make use of MPI communication MPI has to be set up. This means calling `MPI_Init` and `MPI_Finalize` at the right places in the program.
2. **Remove the creation of OpenMP threads from the program**: Replacing OpenMP functionality by MPI functionality means that an OpenMP thread will be replaced by an MPI process. Therefore the creation of threads has to be removed from the program. This can be achieved by removing the call to the OpenMP Runtime Library function `kmpc_fork_call`.
3. **Execute each OpenMP microtask once per process**: Since the removal of OpenMP thread creation will also remove any execution of the code that should be parallelized, direct calls to the microtask functions will have to be added.
4. **MPI communication for shared variables has to be added**: By moving the parallelization of the program from threads into processes the variables of

the program won't be in one shared memory anymore. This means that the modification of a shared variable inside a OpenMP parallel pragma has to be communicated between all processes by using MPI.

The next sections of this chapter will describe how I implemented an LLVM Pass to achieve these steps.

4.2 Structure of the LLVM Pass

The first choice when implementing a custom LLVM Pass is to choose what kind of pass one wants to create. I chose to implement a *ModulePass* because the pass has to inspect the whole program and modify multiple functions.
Listing 4.4 shows the basic structure of the pass. The pass itself is a *struct* that is derived from the *ModulePass* class. The virtual function `runOnModule` from the *ModulePass* class is overridden. This is the starting point of the pass when it is executed by the optimizer.

```
1  using namespace llvm;
2  namespace
3  {
4      struct Omp2mpiPass : public ModulePass
5      {
6          static char ID;
7          Omp2mpiPass() : ModulePass(ID) {}
8          //Pass starts here
9          virtual bool runOnModule(Module &M)
10         {
11             return true;
12         }
13     };
14 }
15 char Omp2mpiPass::ID = 0;
16 static void registerOmp2mpiPass(const PassManagerBuilder &,
       ↪ legacy::PassManagerBase &PM)
17 {
18     PM.add(new Omp2mpiPass());
19 }
20 static RegisterStandardPasses
21     RegisterMyPass(PassManagerBuilder::EP_ModuleOptimizerEarly,
          ↪ registerOmp2mpiPass);
22 static RegisterStandardPasses
23     RegisterMyPass0(PassManagerBuilder::EP_EnabledOnOptLevel0,
          ↪ registerOmp2mpiPass);
```

Listing 4.4: Basic structure of the pass

Lines 15 to 23 are responsible for adding the pass to the LLVM *PassManager*. This method of registering a module pass is implemented after the example given in [gho16]. The *PassManager* is scheduling the order in which passes are executed during the optimization. We can roughly choose at which point in the optimization process our pass is executed by passing `EP_ModuleOptimizerEarly` in line 21. This causes the pass to be executed at the beginning of the optimizer. Other possibilities would be to execute the pass at the end of the optimization process or at specific points, like the vectorization phase. Executing the pass early in the optimization process gives LLVM the chance to further optimize the new code that is generated by our pass. It is also easier to work on unoptimized IR code in the pass, because it is still structured in a very straight forward way. Heavily optimized code can be hard to read for a human, because some optimizations are not intuitive from a humans perspective. In lines 22-23 we make sure the pass is also executed at optimization level zero. This would normally not be the case, since at optimization level zero no optimization passes are executed by LLVM.

4.3 Removing OpenMP

As explained in Section 4.1 all OpenMP operations in IR code are in the form of calls to the LLVM OpenMP Runtime Library. The parallel pragmas are converted to `kmpc_fork_call` function calls, which then create the OpenMP threads and call the microtask function. To remove the creation of threads, the pass first has to identify all instances of this function call.
The `runOnModule` method is the starting point of our pass and it gets passed a reference to the module the pass is working on. To find the OpenMP function calls, the pass loops through all functions of the module. It then checks for every instruction in the functions, whether it is a call instruction with `kmpc_fork_call` as its called function. If the pass finds a function call of this type, it has to replace it with a direct call to the corresponding microtask. This removes all OpenMP parallelization from the program and makes it execute the microtask exactly once.
To achieve this, we can use the `IRBuilder` class, which lets us modify and generate IR code. By constructing an `IRBuilder` object and passing a pointer to the `kmpc_fork_call` call instruction to its constructor we set the insert point of the builder directly in front of the call instruction. We can now delete the call instruction with the `IRBuilder` and create a new instruction of the LLVM type `CallInst`, which calls the microtask function. The `CreateCall` method of the `IRBuilder` class expects a pointer to a `Function` type object and a vector of `Value` pointers as arguments. We can get these by going through the `ArgOperand` list of the `kmpc_fork_call` instruction, provided by the `CallInst` class. As we know from Listing 4.3, the microtask function is the third argument of `kmpc_fork_call`.
Most general functions in the LLVM API are returning or expecting objects of the LLVM type `Value`. This is due to the fact that `Value` is the base class of nearly all LLVM types and this makes it easier to provide general functions to the developer without having to overload them to accept all different kinds of subclasses. On the other hand this means

that developers have to make extensive use of casting if they need an object to be of a specific subtype.
The objects we get from the argument-list of our call instruction are all of the `Value` type, meaning that we have to use a dynamic cast (`dyn_cast`) to transform the third argument into a `Function` type object. After getting a pointer to the correct microtask function, we have to construct a vector of `Value`s , representing the arguments we pass to the microtask function. As stated in Section 4.1 a microtask function expects pointers to `global_tid` and `bound_tid` followed by pointers to all shared variables of the parallel pragma as arguments. The first two arguments are only important for the OpenMP functionality of the microtask and can be replaced by `nullptr`s. The shared variable pointers can be copied from the argument list of the `kmpc_fork_call` call instruction, since it also expects them.
We can now remove the old `kmpc_fork_call` call instruction and replace it with the newly constructed function call to the microtask. This procedure effectively removes all OpenMP parallel pragmas from the code. If the compiled program is now executed it always behaves like it was run with `OMP_NUM_THREADS=1`.

4.4 Adding MPI

After removing OpenMP parallelization, the next step for the pass is adding MPI code. Up to this point all code generation from the pass has been done by adding or removing single instructions of IR code using the `IRBuilder` class. This is not problematic if only a few lines of code have to be modified, but it can lead to confusing and unmaintainable code, when more complex procedures have to be generated. Since LLVM IR is an assembly like language even a few lines of code in a language like *C++* can translate into a huge block of IR instructions. Generating these manually, instruction by instruction, with an `IRBuilder` would produce code that is hard to understand and to modify later on. To circumvent this problem we can use the `IRBuilder` class to create calls to external functions, that are written in *C++*. This gives us the possibility to write complex behaviors, like for example MPI communication schemes in a more human readable form and insert them into the program with one IR instruction. It also makes it easier for us to try out different methods of adding MPI communication to the program, since we can write two different functions that have the same outcome and insert them into our pass by just changing one line of code.
This method is implemented by creating a separate file of source code in which the *C++* functions that will be used by the pass are defined. This code is then compiled into an object file. The object file is then linked together with the object file that is created from the program that the pass is run on. To use the external functions in the pass they have to be declared in our program.

```
1 Module* M = ...
2 LLVMContext& Ctx = ...
3 IRBuilder<> builder(...);
```

```
4 Constant* func = M->getOrInsertFunction("_Z10print_funci",
    ↪ Type::getVoidTy(Ctx), Type::getInt32Ty(Ctx), NULL);
5 Value* arg = builder.getInt32(42);
6 builder.CreateCall(func, arg);
```

Listing 4.5: Example of inserting an external function call into IR code.

Listing 4.5 illustrates this process. To declare the external function we can use the `getOrInsertFunction` method from the `IRBuilder` class. It expects the signature of the external function as arguments and returns a pointer to a `Constant` object.
In line 4 of our example we declare a function with the return type void, that expects an integer as argument. The function's name differs from its source code name, because Clang uses a compiler technique called name mangling [Str88] during the translation of source code into IR.
To get the mangled function name we can inspect the created IR code of the external functions by compiling it with Clang and adding the `-emit-llvm` flag. After the function is declared in the pass it can be called by using the `CreateCall` method of `IRBuilder` (line 6) and passing it the `Constant` pointer followed by the arguments as a vector of `Value` pointers or, in the example's case, a single `Value` pointer.
The first step of adding MPI functionality to a program with our pass is to insert the `MPI_Init` and `MPI_Finalize` function calls to the code. These functions set up MPI at the beginning and do the necessary cleanup at the end of the program. Our pass assumes that the input is a pure OpenMP program, which does not use MPI yet.
To call the MPI functions from our pass we can write simple wrapper functions for them in the external functions source file and declare those in our pass. Adding `MPI_Init` to the beginning of the program is done by searching for the *main* function and inserting a call instruction to `MPI_Init` at the beginning of its *entry* block. The `MPI_Finalize` call instruction is added in front of all exit points (return statements) of the *main* function.

At this state the pass removes OpenMP parallel pragmas and sets up MPI. A program compiled with the pass can be run with MPI and if the parallel sections do not make use of shared variables the MPI version of the program already computes the same results.

4.4.1 Shared variables

In OpenMP parallel sections there is a distinction between two types of variables. Private variables are local variables for each thread. Their values are independent from each other in every thread. A common use case for a private variable in a parallel section is, for example the counter variables in loops. These variables should be local to each thread because in a parallelized loop each thread is responsible for a different part of the loop's range.
Shared variables on the other hand are globally accessible for every thread. If one thread changes the value of a shared variable it is also changed for all other threads. Shared variables are often used to store the result of the parallelized computation. In OpenMP all threads have access to the same shared memory. This makes it rather unproblematic

to handle shared variables since all threads are accessing the same memory location when they access the variable. The OpenMP implementation only has to take care of preventing race conditions when two threads try to access a shared variable simultaneously.
For a code section that is parallelized with MPI instead of OpenMP the variables for each process are not stored in the same shared memory. Every process has its own set of variables, which are stored in the process's local memory that is not visible for any other process. This makes all variables in a code section parallelized by MPI private by default.
To achieve the same behavior for shared variables as in OpenMP in an MPI program, we have to add MPI communication between the processes to synchronize their values. In the pass we distinguish between two kinds of shared variables.
Single value shared variables consist of one element. They are variables of types like *integer* or *char*. When they are modified all of the memory that they allocate is accessed. The other kind of shared variables are arrays. They consist of multiple elements and when one of them is modified only parts of the array's allocated memory is accessed. We distinguish between these types of variables because they have to be treated differently with respect to parallelization. If one process of a parallel computation modifies a single value shared variable it has to be assured that during that time no other process tries to access it. On the other hand, if one process modifies an element of a shared array it is alright for another process to access the other elements of the array at the same time.

To add MPI communication to a program, our pass has to analyze the microtask functions for every parallel section in the code and insert MPI calls at the right places. First it has to check whether any shared variables are used in the microtask. This can be assessed by looking at the number of arguments the microtask function expects. As we know from Section 4.1 a microtask function always has pointers to `global_tid` and `bound_tid` followed by pointers to the shared variables as arguments. If a microtask function only expects two arguments we can be sure that it does not use any shared variables. In this case no modifications to the microtask function are needed, since it only works with private variables, which all variables of an MPI process are by default. If there are more than two arguments in the signature of the microtask function it uses shared variables. The following sections describe how to implement the insertion of MPI communication for single value shared variables and shared arrays.

Single value shared variables

```
1 define internal void @.omp_outlined.(i32* noalias
    ↪ %.global_tid., i32* noalias %.bound_tid., i32*
    ↪ dereferenceable(4) %a, i32* dereferenceable(4) %b) #1 {
2 entry:
3   %.global_tid..addr = alloca i32*, align 8
4   %.bound_tid..addr = alloca i32*, align 8
5   %a.addr = alloca i32*, align 8
6   %b.addr = alloca i32*, align 8
```

```
7   store i32* %.global_tid., i32** %.global_tid..addr, align
      ↪ 8
8   store i32* %.bound_tid., i32** %.bound_tid..addr, align 8
9   store i32* %a, i32** %a.addr, align 8
10  store i32* %b, i32** %b.addr, align 8
11  %0 = load i32*, i32** %a.addr, align 8
12  %1 = load i32*, i32** %b.addr, align 8
13  %2 = load i32, i32* %0, align 4
14  %add = add nsw i32 %2, 1
15  store i32 %add, i32* %0, align 4
16  store i32 42, i32* %1, align 4
17  ret void
18 }
```

Listing 4.6: IR code of a microtask function that uses shared variables

To synchronize the values of shared variables in a microtask function for all processes we first have to track how they are being used in the microtask. Listing 4.6 shows the IR for a microtask function with two shared variables.
In lines 3-6 pointers for every argument of the microtask function are allocated. They point to the memory address of each function argument. This means they are not direct pointers to the values of the arguments, but pointers to their pointers. If there is a load instruction on one of these address variables, the loaded value will be the original pointer to an argument that was passed to the function.
In lines 7-10 the function arguments are stored in their corresponding address variable. This block of instructions will be found at the beginning of every microtask function in LLVM IR.
Lines 11 and 12 are load instructions on the address variables of the two shared variables *%a* and *%b*. These load instructions return pointers to the actual shared variables, which are stored in the two registers *%0* and *%1*. Every time one of the shared variables is used in the microtask function, its value will be accessed by a load instruction on one of these registers. We can see this in lines 13-15, where the value of the shared variable *%a* is loaded from register *%0* and then increased by one in the next line. After the value of *%a* has been modified its new value is stored again at the address pointed to by *%0*, which is the pointer to *%a*.
This pattern of a load instruction on the shared variable followed by a modification of its value and ending in a store instruction is what our pass has to locate in the microtask function. These instruction blocks are changing the value of a shared variable and therefore have to be fitted with MPI functionality to keep the shared variable's value synchronized throughout all MPI processes.
To implement this in our pass, the following steps have to be taken.

1. First we have to locate the previously discussed load instructions, which load direct pointers to the memory location of the shared variables. This is implemented by looping over the `ArgumentList` of the microtask function. For each argument we

get a list of its `User`s. The `users` function is a method of every LLVM `Value` object. It returns an `iterator_range` consisting of every `User` of the `Value`. An `User` of a `Value` is every instruction or operator in the code that has the `Value` as one of its operands.
In our example this gives us the `store` instructions, where the addresses of the arguments are stored in their corresponding address variables (Listing 4.6 lines 7-10). If we again loop over those `store` instruction's `User`s, we find the `load` instructions we were looking for (Listing 4.6 lines 11-12).
Pointers to these instructions are stored by the pass and used as a basis for tracking the uses of shared variables inside the microtask function.

2. After identifying the registers that hold the pointers to the microtask function's shared variables the next step of the pass is, to locate all parts of the function, where a shared variable is modified.
   This is achieved by, again, looping over the `User`s of each of the load instructions. By filtering those `User`s for instructions of the type `LoadInst`, we can get all places in the body of the microtask function, where the value of a shared variable is loaded from its pointer. This indicates that the shared variable will be used in the following lines of code.
   We now have to find out if there is also a store instruction on the shared variable following the load. This would mean that its value has indeed been modified and not just inspected. This is implemented in the pass by inspecting all instructions following the load until either a store instruction or a return instruction is found. In case the pass only finds a return instruction, it means that the value of the shared variable has only been read and therefore does not need to be synchronized for all MPI processes. If the pass finds a store instruction on the same shared variable first, it means that the variable has been modified and the insertion of MPI code is necessary to message its new value to all other processes. If a correct store instruction is found, the pass saves the block of instructions from the load to the store in a *C++* vector.
   The pass does this procedure for all shared variables and produces a vector containing all blocks of instructions that need to have MPI communication added to them.

3. The last step of the pass is the insertion of MPI code for each collected block of instructions. To achieve correct program behavior for the parallel modification of shared variables I decided to implement a simple MPI communication scheme for single value shared variables. It uses blocking MPI operations and serializes the modification operation on the shared variable for all processes. Listing 4.7 shows the implementation of this communication scheme.

```
1 void update_var_before(int rank, int size, void* var,
    ↪ int datatype){
2   if(rank != 0){
```

```
3       MPI_Recv(var, 1, datatype, rank-1, 0,
            ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
4     }
5   }
6   void update_var_after(int rank, int size, void* var,
        ↪ int datatype){
7     if(rank == 0){
8       MPI_Send(var, 1, datatype, rank+1, 0,
            ↪ MPI_COMM_WORLD);
9       MPI_Recv(var, 1, datatype, size-1, 0,
            ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
10    }
11    else{
12      if(rank != size-1){
13        MPI_Send(var, 1, datatype, rank+1, 0,
              ↪ MPI_COMM_WORLD);
14      }
15      else{
16        MPI_Send(var, 1, datatype, 0, 0, MPI_COMM_WORLD);
17      }
18    }
19    MPI_Bcast(var, 1, datatype, 0, MPI_COMM_WORLD);
20  }
```

Listing 4.7: External C++ functions for synchronizing the value change of a shared variable with blocking MPI operations

It consists of two functions, of which one is placed right in front of the load instruction on the shared variables value and the other after the store instruction. The first function (lines 1-5) stops all processes except rank 0 with a blocking `MPI_Recv` operation. This makes all processes wait to load the value of the shared variable, until it has been successfully modified by the predecessor process.
The second external function (lines 6-20) gets inserted into the microtask function's code right after the store instruction on the shared variable is called. This function sends the value of the shared variable after it has been modified to the next process. The last process sends the final value of the shared variable back to the rank 0 process, which is also made to wait with the blocking receive instruction in line 9. After the final value of the shared variable has been computed and send back to rank 0, the function calls a `MPI_Bcast` operation, which broadcasts the final value to all processes. After this the program flow is continued by all processes.
Finally the pass adds these two function calls to all blocks of load/store instructions that have been identified in the microtask. The function is now successfully translated to an MPI parallelized version for single value shared variables.

The MPI communication scheme that has been added here only works on a limited

subset of all possible microtask functions. It has the assumption that every modification on a shared variable is executed by all running processes. If a process does not participate, the program will run into a deadlock. This means that modifications on shared variables inside branches of the function, which are not reached by all processes during execution, are not yet supported by the pass. To make the pass work with all possible microtask functions more work has to be put into it, to cover all possible edge cases. Different MPI communication schemes, that do not rely on every process's participation need to be added.

Shared Arrays

Parallelizing the use of shared array variables inside a microtask function is implemented differently to the procedure described in the last section. I decided on using MPI-RMA operations to implement the parallel access on elements of a shared array.

```
1 %0 = load [10 x i32]*, [10 x i32]** %a.addr, align 8
2 %arrayidx = getelementptr inbounds [10 x i32], [10 x i32]*
    ↪ %0, i64 0, i64 0
3 %1 = load i32, i32* %arrayidx, align 16
4 %add = add nsw i32 %1, 1
5 store i32 %add, i32* %arrayidx, align 16
```

Listing 4.8: Example IR code of accessing the elements of an array inside a microtask function

Listing 4.8 shows parts of a microtask function, where an element of a shared array is accessed and modified.
In line 1 we see the load instruction on the address variable of the microtask's shared array argument. Line 2 introduces the `getelementptr` instruction. We know from Listing 4.6 that the value of a shared variable inside a microtask function is accessed by load instructions on a pointer to it. Since arrays are multi-element objects we have to compute the right memory address before we can access one of their elements.
In LLVM IR this is done through the `getelementptr` instruction, which returns a pointer to the address of an array or struct element. In line 2 of the example a pointer to the first element ([0]) is created. The actual value of the array element is then loaded in line 3, by passing the element pointer to a load instruction. Next the loaded value is increased by one and the new value is stored at the same location in the array's body.
To parallelize a microtask function with shared arrays with our pass the following steps have to be implemented.

1. The first step is identical to the procedure for single value shared variables. We have to locate the load instructions on the address variables for all shared variables of the microtask function. This produces a *C++* vector of load instructions like *%0* from Listing 4.8.

2. The Next step is to create `MPI_Window` objects for every shared array. To be able to use MPI-RMA operations for accessing the shared array each process needs to declare the creation of a window with the same size of allocated memory as the array.

```
1 void shared_array_window_creation(void* array, int
    ↪ array_size, MPI_Win* win, int datatype){
2   int size;
3   MPI_Type_size(datatype, &size);
4   MPI_Win_create(array, array_size, size,
      ↪ MPI_INFO_NULL, MPI_COMM_WORLD, win);
5 }
```

Listing 4.9: External function for creating MPI_Window for shared array

The windows are created by inserting a call to the external function shown in Listing 4.9 into the code, right after the load instructions for the shared array pointers.
The first two arguments of the function are a void pointer to the base address of the array and the number of array elements. The third argument is a pointer to an `MPI_Window` object, in which the created window will be stored. The last argument is an `MPI_Datatype` object, which declares the type of the array. The two MPI types `MPI_Window` and `MPI_Datatype` are represented as integers when they get translated from *C++* to LLVM IR.
The function creates a new `MPI_Window` and stores it in the pointer given to as its third argument. The pointer to the window and the pointer to the shared array get stored by the pass in a *C++* map container. This is an easy way to always identify which window belongs to which array if there are multiple shared arrays in the microtask function.

3. After the windows for all arrays have been created we need to identify all places in the microtask function, where a shared array is accessed. This is implemented by going through all *User*s of the shared array pointers. If the *User* is a `getelementptr` instruction it means that an element of the array will be loaded from or stored to. The pass stores all found `getelementptr` instructions in a *C++* vector.

4. Now that we have the locations of all accesses to a shared array we can add MPI communication to them.
The MPI communication scheme I implemented for shared arrays is structured in the following way:
The process with the rank 0 always holds the "right" version of the array. This means that all other processes use MPI-RMA operations to update the array in the rank 0 process. If there is a load to an element of the array in another process, the process calls `MPI_Get` and updates the value of its local array with the value from rank 0. If there is a store to an element of the array, the other processes call `MPI_Put` and store the changed value in the array local to rank 0. This causes that

at the end of the microtask function only rank 0 has the correct array. This is fixed by letting all other processes copy all values from the rank 0 array at the end of the microtask function, before the `MPI_Window` is freed. This communication scheme is implemented in the form of three external functions (see Listing 4.10), getting inserted into the pass at the right places.

```
1  void store_array(void* array, int displacement, int
     ↪ rank, MPI_Win* win, int datatype) {
2    if(rank != 0) {
3      MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 0, 0, *win);
4      MPI_Put(array, 1, datatype, 0, displacement, 1,
         ↪ datatype, *win);
5      MPI_Win_unlock(0, *win);
6    }
7  }
8  void load_array_value(void* array, int rank, int
     ↪ displacement, MPI_Win* win, int datatype) {
9    MPI_Win_lock(MPI_LOCK_SHARED, 0, 0, *win);
10   MPI_Get(array, 1, datatype, 0, displacement, 1,
       ↪ datatype, *win);
11   MPI_Win_unlock(0, *win);
12 }
13 void synch_and_free_array(void* array, int array_size,
     ↪ int rank, MPI_Win* win, int datatype) {
14   MPI_Win_fence(0, *win);
15   if(rank != 0) {
16     MPI_Get(array, array_size, datatype, 0, 0,
         ↪ array_size, datatype, *win);
17   }
18   MPI_Win_fence(0, *win);
19   MPI_Barrier(MPI_COMM_WORLD);
20   MPI_Win_free(win);
21 }
```

Listing 4.10: External functions for parallelizing microtask functions with shared arrays

Function one (lines 1-7) is responsible for writes to the array. If the function is executed by a process, not having the rank 0, it calls an MPI lock on the array's window in process 0. It then uses the `MPI_Put` operation to write the value of the local array element at the index given in the `displacement` argument to the array from process 0. A call to this function is inserted by the pass directly after store instructions to the array. Through this, the array of process 0 always gets updated when another process modifies an element.
Function two (lines 8-12) is the counterpart to function one and is responsible for

loading the value of an array element from the rank 0 process. The pass adds a call to this function in front of all load instructions to the array. This way all processes load the "correct" values from the array.
The last function (lines 13-21) frees the `MPI_Window` in line 20, after it lets all processes copy the whole contents of the array from process 0. This is inserted into the microtask function in front of every return statement. It causes every process to have the same version of the array as process 0 after leaving a parallelized section.

The MPI communication inserted into the microtask functions here, successfully parallelizes them. However it does not guarantee the expected result of a parallel section if there is the possibility of modifications to the same array element at the same time. For example a microtask function, where the same array element is modified by all processes simultaneously does not always produce the expected output. This however is also not guaranteed in the original OpenMP version of the parallel section. To ensure that the array element can only be modified by one OpenMP thread at the same time, synchronization pragmas like `omp critical` would be necessary. Since our pass does not support this feature yet it is assumed here that they are not used in the original OpenMP program.
The concrete implementation of this procedure in the pass also only works under limited circumstances at the moment. It expects an array that is allocated on the *stack* and has a constant size (e.g. `int arr[10];`). This is due to the fact that LLVM IR only recognizes an array in the IR form of `[N x type]` when it was created under these circumstances. If it gets created with a variable size or through the **`new`** keyword on the heap, the array variable is treated as a normal pointer object in LLVM IR. The information about its size is carried through the IR code separately. This means that more work needs to be done on the pass implementation to make it able to handle these cases, because the size of the array has to be known in the microtask function to create an `MPI_Window` for it. The described technique to synchronize the array with MPI would not have to change to make it work for these types of arrays, but more supporting code has to be developed for recognizing the pointer as an array and for figuring out its size.

4.5 Replacing special OpenMP functions

Additionally to the preprocessor directives, the OpenMP API also provides a variety of OpenMP functions that can be called in an OpenMP program. The two functions `omp_get_thread_num` and `omp_get_num_threads` are probably the most often used ones. They return the number of the current thread and the overall number of running threads inside an OpenMP parallel section. They are both essential, when you want the threads in a parallel section to execute different parts of the code. For example, a parallel pragma in which each thread is supposed to write to a different part of a shared array, needs to have information about the assigned number of every thread.
When we want to translate OpenMP programs into MPI programs, these functions also have to be translated. MPI provides exact counterparts to the two mentioned

OpenMP functions. `MPI_Comm_rank` returns the number of the current process's rank and `MPI_Comm_size` returns the overall number of MPI processes. Since these two MPI functions have the same output as their OpenMP versions, our pass simply has to replace each function call to `omg_get_num_threads` and `omp_get_thread_num` with `MPI_Comm_size` and `MPI_Comm_rank` function calls.

```
1 %call = invoke i32 @omp_get_thread_num()
2         to label %invoke.cont unwind label %terminate.lpad
3 invoke.cont:           ; preds = %entry
4   store i32 %call, i32* %rank, align 4
5   %call2 = invoke i32 @omp_get_num_threads()
6           to label %invoke.cont1 unwind label %terminate.lpad
7 invoke.cont1:          ; preds = %invoke.cont
8   store i32 %call2, i32* %size, align 4
```

Listing 4.11: IR code of a call to OpenMP functions

Listing 4.11 shows parts of a microtask function, in which the two variables *rank* and *size* are assigned the return values of `omp_get_thread_num` and `omp_get_num_threads` call instructions.
In line 1 we can see the call to `omp_get_thread_num`. Instead of the `call` keyword to call the function, the `invoke` keyword is used here. `invoke` is used for function calls, where the callee function might throw an exception. As we can see in line 2 of the example, the control flow of the program gets influenced by the invoke instruction. If the called function does not throw an exception the program jumps to the label `invoke.cont`. If however the function does throw an exception, the program jumps to the `terminate.lpad` label, which handles the exception. In lines 5-6 we can see the same procedure for a call to `omp_get_num_threads`.
To translate the shown IR code into an MPI version we have to replace the called functions in the invoke instructions with calls to their MPI counterparts. Our pass implements this in form of a function, which searches the whole module for calls or invokes to OpenMP functions. It then modifies the instructions directly and replaces the called function with a call to external wrapper functions for `MPI_Comm_rank` and `MPI_Comm_size`.

```
1 if(Function* omp_get_thread_num =
     ↪ M.getFunction(StringRef("omp_get_thread_num"))) {
2   std::vector<Value*> function_calls;
3   for(auto* user : omp_get_thread_num->users()) {
       ↪ function_calls.push_back(user); }
4   for(auto* instruction : function_calls){
5     if(auto* invoke = dyn_cast<InvokeInst>(instruction)){
6       invoke->setCalledFunction(mpi_rank_func);
7     }
8   }
```

```
9 }
```

Listing 4.12: Replacing of OpenMP function calls with MPI function calls

Listing 4.12 shows parts of the *replace* function. In line 1 we use the `Module` class's member function `getFunction` to claim a pointer to the function object for `omp_get_thread_num`. We then loop over all `Users` of the function (lines 2-3), to get a vector of all instructions in the module that use the function.
In lines 4-5 we check for every collected `User` object whether it is an invoke instruction. If it is one, we use the `setCalledFunction` member function of the `InvokeInst` class to switch the call to `omp_get_thread_num` with a call to our external `mpi_rank_func`, which is a wrapper around `MPI_Comm_rank`.
The same procedure is also done for `omp_get_num_threads`. When the pass uses this *replace* function all calls in the program to the two discussed OpenMP functions are successfully replaced by their MPI counterparts.
The OpenMP API provides more than the two discussed functions, but they are not yet supported by our pass.

4.6 The finished Pass

```
1  struct Omp2mpiPass : public ModulePass {
2    //Variables for all external functions used in the pass
3    Constant* external_func1 = nullptr;
4    ...
5
6    //Memberfunctions of the Pass
7    //Declares all external functions that can be used in the
       ↪ pass
8    void load_external_functions(Module &M);
9    //Replaces OpenMP special Functions
10   void replace_omp_functions(Module &M);
11   //Inserts MPI Initialization and Finalization
12   void setup_mpi(Function &F);
13   //Adds MPI to single value shared variable microtask
       ↪ functions
14   void shared_parallel(Function* microtask, Module &M);
15   //Adds MPI to microtask functions with shared arrays
16   void shared_parallel_arrays(Function* microtask, Module
       ↪ &M);
17   //"main" function of the pass
18   virtual bool runOnModule(Module &M);
19 }
```

Listing 4.13: Interface of the finished pass

Listing 4.13 shows the interface of the LLVM Pass that has been developed in this thesis. This section will give a brief overview of its final structure and a few pseudo code examples for functions, that have not been discussed yet.
At the top of the pass (lines 2-4), variables for all external functions that can be used in the pass are declared. They are attributes of the pass, so every member function can have access to them.
Lines 6-18 of Listing 4.13 display the declarations of all member functions.

1. `load_external_functions` (line 8) is responsible for initializing the external function variables of the pass. It uses the `getOrInsertFunction` function of the `Module` class to create LLVM `Function` objects for every external function and assigns them to the variables declared in lines 2-4.

2. `replace_omp_functions` (line 10) is the function that was discussed in Section 4.5. It replaces all call instructions to special OpenMP functions in the module by their MPI counterparts.

3. The `setup_mpi` function (line 12) inserts the `MPI_Init` and `MPI_Finalize` operations into the program. It gets passed a pointer to the *main* function of the program and adds MPI initialization at its beginning and MPI finalization in front of all exit points of the function.

4. The two functions declared in lines 14 and 16 add the MPI communication to the program. They do the majority of the work in the pass and their implementation has been discussed in Section 4.4.1.

5. The `runOnModule` function (line 18) is an overridden version of the virtual `runOnModule` member function from the `ModulePass` class, from which our pass is derived. It can be seen as the "main" function of our pass.
   It is responsible for locating all OpenMP operations in the program and extracting their microtask functions. The next section will give a short overview of the implemented `runOnModule` function.

4.6.1 The runOnModule function

```
1  bool runOnModule(Module &M):
2    load_external_functions(M);
3    replace_omp_functions(M);
4
5    for all Functions F in M:
6      if(Function is "main"):
7        setup_mpi(F);
8
9      for all Instructions I in F:
10       if(I is Callinstruction to "__kmpc_fork_call"):
```

```
11          microtask = I.getArgOperand(2);
12
13          if(microtask has shared single value variables):
14            shared_parallel(microtask, M);
15          if(microtask has shared array variables):
16            shared_parallel_arrays(microtask, M);
17
18          args = arguments for microtask;
19          builder = IRBuilder object;
20          builder.setInsertPoint(I);
21          builder.CreateCall(microtask, args);
22          I.eraseFromParent();
```

Listing 4.14: Pseudo Code version of the runOnModule function

Listing 4.14 shows a pseudo code version of the `runOnModule` function, implemented in the pass. It has been stripped of many details, like the necessary casting operations on the LLVM `Value` objects and only illustrates the important steps of the function.
In lines 2 and 3 the external function variables of the pass are initialized and all special OpenMP functions are replaced. After that the pass loops over all functions of the module and first searches for the *main* function. In line 7 MPI is set up by calling the `setup_mpi` function and passing it a pointer to the *main* function.
The next step is to locate all calls to the `kmpc_fork_call` function, which represents the execution of an OpenMP parallel section. This is done by filtering all instructions in the program for a call instruction to `kmpc_fork_call` (line 10).
When a correct call instruction is found, the pass has to extract the microtask function, that is called by the fork call. This is implemented in line 11, where the third argument of the call instruction is extracted. As we know from Listing 4.3, the third argument of a fork call is always the microtask function.
After the right microtask function has been identified, we inspect its arguments and check for shared variables. Depending on the types of the shared variables of the microtask, the correct functions for adding MPI communication to it are called by the pass (lines 13-16).
After the microtask function has been equipped with MPI operations, the last step of the `runOnModule` function is to replace the `kmpc_fork_call` call instruction with a direct call to its microtask function. This is done in lines 18-22. First the list of arguments for a microtask function call is created. Then, using the `IRBuilder` class, a new call instruction is inserted into the program, right in front of the fork call. At last the old call instruction is removed.

5 Related Work

There has been a lot of research about the comparison of *OpenMP* to *MPI*. This research is mostly concerned with a direct performance comparison.
The results of these studies vary. In [Kra03] the authors compared different *OpenMP* implementations of of a subset of the NAS benchmark with its *MPI* version. They came to the conclusion that *OpenMP* can provide competitive performance results to *MPI*, but it requires a very sophisticated *OpenMP* program style, which is not common in every *OpenMP* program. In [JJF+99] another *OpenMP* implementation of the NAS benchmark was compared to the original *MPI* version. The authors concluded that *OpenMP* can deliver competitive results to *MPI* but *MPI* provides better scalability.
These findings show that on a shared memory system the performance of *OpenMP* and *MPI* can be very close to each other, but *MPI* still has the advantage of better scalability and it works on distributed memory systems as well.
[Squ16] focuses on the replacement of *OpenMP* with *MPI-RMA*. The author explored different approaches of replacing *OpenMP* code with *MPI-RMA* operations and showed that under the right circumstances, *MPI-RMA* versions of a program can even outperform the original *OpenMP* version.
The approaches used in [Squ16] could be very helpful for the further development of our LLVM pass. However, this requires deeper analysis of the *OpenMP* sections that are translated by the pass, to select the right *MPI* communication scheme, that fits the section best.
All of the above mentioned research has in common that the translation from *OpenMP* to *MPI* or vice versa, was implemented manually by a programmer. The automatic translation between the two parallelization approaches does not seem to have much research on it.
[BE05] discusses a compiler driven approach for source-to-source translation of *OpenMP* to *MPI*. It follows a similar approach to this thesis, by replicating shared data to every *MPI* process and also executing the serial parts of the program by all processes. The code translation is mainly focused on parallelized *OpenMP* loops. The discussed tool divides the loop's work on all processes and computes an *array-dataflow* analysis to identify the needed *MPI* communication.
Even though the paper discusses a source-to-source translation of *OpenMP* to *MPI*, instead of the translation during compilation to a binary, which this thesis discusses, the methodology could be interesting for our LLVM pass. The *array-dataflow* analysis approach to determine the insertion of *MPI* communication, could be very beneficial to implement in our pass.
Overall it is safe to say that the automatic translation of *OpenMP* to *MPI* has not been fully explored yet and much interesting work can still be done in the future.

6 Results

The LLVM pass for translating OpenMP into MPI, which has been developed in this thesis is able to transform a limited subset of OpenMP programs into MPI programs successfully. Following is a list of features the final pass has to offer.

1. **pragma omp parallel**: The pass can translate simple `omp parallel` sections into MPI code. So far the pass does not support other kinds of OpenMP pragmas like `omp parallel for`. All `omp parallel` pragmas, that do not use shared variables can be fully translated. A subset of all `omp parallel` pragmas with shared variable usage can be translated.

2. **Shared variables**: The pass is able to translate specific parallel pragmas with shared variables. It supports single value shared variables in parallel sections, as long as each modification to the shared variables is executed by all threads/processes in the parallel section.
   Shared arrays, that are allocated on the stack with a fixed size at compile time are also supported. Simultaneous writes on an array element by multiple processes at the same time leads to undefined results. But this can also be the case in the original OpenMP version of the parallel section.

3. **OpenMP special functions**: The two OpenMP functions `omp_get_thread_num` and `omp_get_num_threads` are translated into equivalent MPI operations by the pass.

4. The pass also allows future developers to change the MPI communication schemes, used for shared variables, without having to rewrite big parts of the pass. The method of defining the specific MPI operations in external functions written in *C++* allows the testing of different communication schemes without much work.

Even though the implemented features only cover parts of *OpenMP* yet, the pass is a good foundation for future work. This thesis discussed the structure of IR code for *OpenMP* programs, how to locate OpenMP parallel sections in it and how to modify them. The basic methods developed here, can be applied to the implementation of new features, that cover the remaining *OpenMP* operations, which are not yet supported by the pass.
The *MPI* programs produced by the pass offer greater scalability in contrast to the original *OpenMP* programs. They can be executed over multiple nodes of a distributed memory system. The achieved performance of the translated programs is not optimal yet and they will, most of the time, run slower than their *OpenMP* counterparts, if

executed with the same number of threads/processes. This is something that needs to be looked at. *MPI* communication schemes that are more specific to the behavior of the transformed *OpenMP* parallel section have to be implemented.
Overall the thesis provides a successful *proof-of-concept* for the automatic translation of *OpenMP* to *MPI*, but more work needs to be done to achieve a complete coverage for all of *OpenMP*'s features.

6.1 A benchmark example

To test the functionality and performance of the developed pass, I implemented a small *OpenMP* program, which mimics the data access pattern of a partial differential equation solver using the Gauß-Seidel method. The program loops over a matrix several times and computes the average value of each element's neighbors. This is repeated for a chosen number of iterations.

```
1  while(iterations > 0){
2    ...
3    #pragma omp parallel private(star) shared(matrix) {
4      int rank = omp_get_thread_num();
5      int size = omp_get_num_threads();
6      int lower = rank * ( (width-2) / size) + 1;
7      int upper = (rank+1) * ( (width-2) / size) + 1;
8
9      for(int i = lower; i < upper; i++){
10       for(int j = 1; j < width-1; j++){
11         star = 0.25 * (matrix[compute_index(i-1,j,width)] +
12             matrix[compute_index(i+1,j,width)] +
13             matrix[compute_index(i,j-1,width)] +
14             matrix[compute_index(i,j+1,width)]);
15         matrix[compute_index(i,j,width)] = star;
16         }
17       }
18     }
19     iterations--;
20   }
```

Listing 6.1: Benchmark program written with OpenMP.

The main loop of the program is shown in Listing 6.1. The outer for loop is parallelized with an `omp parallel` pragma. This could be done more effectively with an `omp parallel for` pragma, which is not yet supported by the pass.
The range of the for loop in line 9 is distributed between all threads. Since this example only exists to test our pass, no meaningful result is computed here and the matrix is only later used to verify that the *OpenMP* and the translated *MPI* version produce the same output.

# Threads/Processes	OpenMP	MPI
Arraysize = 102, Iterations = 100	-	-
4	0.020s	2.239s
10	0.025s	2.139s
Arraysize = 1002, Iterations = 100	-	-
4	1.181s	214.807s
10	0.616s	205.052s
Arraysize = 1002, Iterations = 5	-	-
4	0.115s	10.58s
10	0.091s	10.585s

Table 6.1: Results of comparing the performance of the OpenMP and the translated MPI version of the example program.

The program was run multiple times on one computing node, using a *Intel Xeon X5650* processor. The operating system of the system is *Ubuntu 16.04.3 LTS*.
Table 6.1 shows the results of executing the program in its *OpenMP* form and after being compiled with our translation pass. The execution time for different sizes of the matrix and different numbers of iterations has been measured. The table contains the average execution time of the program under different settings. Each measurement was repeated three times. All comparisons are between the same number of *OpenMP* threads and *MPI* processes.
The first observation was that the output of both program versions is identical for every run. This means that the semantics of the program have been preserved after its translation into *MPI*.
The results of the benchmark quite obviously tell that the translated *MPI* version delivers a severely worse performance in comparison to the original *OpenMP* version. The original program runs about 100 to 200 times faster than the translated version. This likely results from the implemented *MPI* communication scheme for shared arrays. There is severe overhead due to the fact that each process contains the whole array and copies it multiple times at the end of a microtask function to let every process have the same version.
More work has to be done on the pass to implement smarter communication schemes for shared variables. This will be discussed further in Chapter 8.

7 Conclusion

In conclusion we can say that this thesis provides a successful *proof-of-concept* for the automatic translation of *OpenMP* to *MPI*. We saw that the LLVM infrastructure offers a good framework for automated code transformation and that LLVM IR code delivers all the needed information about *OpenMP* code for translating it. Due to the low level representation of source code in the IR form, many details have to be taken into consideration when working on it, but on the other hand the developed pass has a good generality. It will be able to translate code from any programming language that is supported by *OpenMP* and has an LLVM front-end.
The pass is not able to translate every *OpenMP* program yet, but additional features can be added, by using the basic methods described in this thesis, to eventually cover all of *OpenMP*.
From a performance perspective, the implemented pass does not yet deliver a realistic replacement for *OpenMP*. More work is required to implement performant *MPI* communication schemes, that get closer to the original *OpenMP* performance of the program. But this was not the main focus of the thesis.
Even though the performance of the produced *MPI* programs is not good yet, their scalability is already improved and the program is no longer limited to be used on shared memory systems.
The compiler based approach for automatic translation looks very promising and the developed LLVM pass provides a solid foundation for future work.

8 Future work

Since the developed LLVM pass does not support all *OpenMP* features yet, there are multiple things that can be improved on in the future.

- The supported kinds of shared variables in parallel sections can be expanded on. Support for shared arrays with variable size and shared arrays, that are allocated on the *heap* needs to be implemented. The challenge with these types of arrays is that they are not clearly identified as arrays in IR code. They are represented in the form of normal pointers instead of the `[N x type]` format. This means that the pass needs to identify that they are pointers to the beginning of an array and it needs to figure out the arrays size. This information is not carried with the array pointer variable and needs to be located by the pass.

- The `omp parallel for` pragma is provided by *OpenMP* for the automatic parallelization of `for` loops. It is one of the most used *OpenMP* features in actual programs. The pass can be extended to also support this feature. The successful implementation of this and the previous point, would probably make the pass able to fully translate many "real world" *OpenMP* programs.

- The performance of the *MPI* programs produced by the pass is not good at the moment. As we saw in Section 6.1 the original *OpenMP* versions outperform their translated *MPI* versions severely. This stems from the very general *MPI* communication schemes that are inserted into the code to replace *OpenMP*.
  The implementation of more performant *MPI* routines for shared arrays is a important step to make the pass usable in a real world environment. An approach, which distributes the data of an array between all processes instead of having complete copies of the array in all of them seems likely to give a big performance gain.
  Also transforming the whole program into a state, where only one process computes the non-parallel sections of the code would remove current problems of the pass. In that case only one process would need the final version of the shared array, produced by a parallel section. This would eliminate the necessity of copying the array from rank 0 to all other processes at the end of a microtask function, which takes a lot of time and is probably a main reason for the bad performance of the created *MPI* programs.
  In general, more specific communication schemes for different kinds of parallel sections can be added. However this would need a deeper analysis of each microtask function to determine a fitting communication scheme.

- More special *OpenMP* functions and pragmas can be added to the pass. With more development work, the pass could be expanded to cover the whole feature set of *OpenMP*. This would make it possible to transform any *OpenMP* software into an *MPI* version, that can be run on multiple nodes of a HPC cluster.

Bibliography

[AB12] Greg Wilson Amy Brown. *The Architecture of Open Source Appllications*, volume I. 2012.

[BE05] Ayon Basumallik and Rudolf Eigenmann. Towards automatic translation of openmp to mpi. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 189–198, New York, NY, USA, 2005. ACM.

[gho16] ghost. Module pass with auto registration. https://github.com/sampsyo/llvm-pass-skeleton/issues/7, 2016.

[JJF+99] H. Jin, H. Jin, M. Frumkin, M. Frumkin, J. Yan, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. Technical report, 1999.

[Kra03] Géraud Krawezik. Performance comparison of mpi and three openmp programming styles on shared memory multiprocessors. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '03, pages 118–127, New York, NY, USA, 2003. ACM.

[MPI17] MPI Forum. Mpi 4.0. `http://mpi-forum.org/mpi-40/`, 2017.

[Ope17] OpenMP. Openmp. `http://www.openmp.org/`, 2017.

[Pro17a] The LLVM Project. The llvm compiler infrastructure. `http://llvm.org/`, 2017.

[Pro17b] The LLVM Project. Llvm doxygen. `http://llvm.org/doxygen/`, 2017.

[Pro17c] The LLVM Project. LLVM Language Reference Manual. `https://llvm.org/docs/LangRef.html`, 2017.

[Pro17d] The LLVM Project. LLVM OpenMP Runtime Library. `https://openmp.llvm.org/Reference.pdf`, 2017.

[Pro17e] The LLVM Project. Llvm tutorial. `https://llvm.org/docs/tutorial/`, 2017.

[Sam15] Adrian Sampson. Llvm for grad students. `https://www.cs.cornell.edu/~asampson/blog/llvm.html`, 8 2015.

[Squ16] Jannek Squar. Mpi-3 algorithms for 3d radiative transfer on intel xeion phi coprocessors. Master's thesis, Universität Hamburg, October 2016.

[Str88] Bjarne Stroustrup. *Type-safe Linkage for C++*. 1988.

[Sut] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software.

List of Figures

List of Listings

List of Tables

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum	Unterschrift

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Ort, Datum	Unterschrift