Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

M A S T E R T H E S I S

Characterization and translation of OpenMP use cases to MPI using LLVM

vorgelegt von
Tim Jammer

MIN-Fakultät
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6527284
E-Mail-Adresse:	3jammer@informatik.uni-hamburg.de

Erstgutachter:	Prof. Dr. Thomas Ludwig
Zweitgutachter:	Dr. Michael Kuhn

Betreuer:	Jannek Squar
	Dr. Michael Kuhn

Hamburg, den 4.12.2018

Abstract

OpenMP makes it fairly easy to program parallel applications. But OpenMP is limited to shared memory systems. Therefore this thesis will explore the possibility to translate OpenMP to MPI by using the LLVM compiler infrastructure. Translating OpenMP to MPI would allow to further scale up parallel OpenMP applications as distributed memory systems may be used as well.

This thesis will explore the benefits of the translation to MPI for several classes of parallel applications, that are characterized by similarity regarding computation and data movement.
The improved scalability of MPI can be exploited best, if there is a regular communication, like a stencil code. For other cases, like a Branch and Bound algorithm, the performance of the translated program looks promising but further tuning is required in order to be able to fully exploit more CPU cores offered by scaling up to a distributed memory system. However the developed translation does not work well when all to all communication is required, like in a butterfly communication scheme of a fast Fourier transform.

Contents

1. Introduction

1.1. Motivation

Parallel programming is a very important topic, as many of todays applications use parallel processing. OpenMP [Ope13] is one of the most important APIs to program a parallel application, as it is fairly easy to use. OpenMP is a portable framework that is in use in many areas, from applications in high performance computing (HPC) or desktop environment to mobile android devices [VSGS13, Ind].
One of the main advantages of OpenMP compared with other techniques of parallelization such as MPI or Pthreads is that OpenMP enables an easy parallelization of an existing program. When parallelizing an application with OpenMP, nearly no rewriting of the serial code has to be done.
One of the main disadvantages of OpenMP is that it is limited to one shared memory system. Therefore a compiler assisted translation of OpenMP to MPI code will allow more scalability of OpenMP programs, so that they can be used on (possibly large scale) distributed memory systems as well. A translation done by the compiler has the advantage, that OpenMP programs do not have to be ported manually to MPI, which would require much effort to rewrite the code.

This limited scalability of OpenMP might present a challenge in the foreeseable future. Interestingly one may see this challenge analogous to the time when the clock rates of processors were not speeding up anymore and a shift to multiprocessing had to be done: "Writing programs that scale with increasing numbers of cores should be as easy as writing programs for sequential computers" [ABD+09]. As multiprocessing was a new concept, it was difficult to learn and use for the users. But OpenMP met this challenge quite well.
Now compiler assisted translation from OpenMP to MPI can make an important contribution in order to face the limited scalability of OpenMP in the future. This would be a valuable contribution to easily exploit the capabilities of distributed memory systems, without the need to learn a new technology.

Therefore this thesis will explore how different use cases of OpenMP can be translated to MPI using the LLVM compiler infrastructure. The goal is to automatically translate an OpenMP program to MPI without the need to rewrite its code.

1.2. Structure of this thesis

First, Chapter 2 gives an overview of the technology used within this thesis (LLVM, MPI and OpenMP).
In Chapter 3 different cases of parallel applications are presented, which were used in this thesis to evaluate the possibility of automatic translation to MPI for different applications.
Chapter 4 presents different MPI communication patterns that will be used for the translation of shared variables to a distributed memory system.
The translation process is described in Chapter 5.
Chapter 6 evaluates the performance and memory usage of the translated programs for different use cases.
This thesis concludes with a discussion of related work in Chapter 7 followed by the summary and interesting points for further research in Chapter 8.

Additionally, the appendix covers some more technical aspects of the translation process (Appendix B) as well as the details of the particular OpenMP cases translated (Appendix A).

2. Overview of the used technology

This chapter explains the technology used in my thesis, namely LLVM (Section 2.1), MPI (Section 2.2) and OpenMP (Section 2.3).

2.1. LLVM

This section provides an overview of the LLVM project. First an overview of LLVM (based on [Ble17]) is given, followed by the explanation of the LLVM intermediate representation (Section 2.1.1) and its comparision to x86 assembler (Section 2.1.2). Also the LLVM and Clang API used in this thesis are described (Sections 2.1.3 and 2.1.4).

LLVM is a compiler infrastructure project written in C++ [llvc]. It started out as a research project at the University of Illinois [LA04]. Since then it has become a big collection of compiler tools, which are being used in open source projects [wit18a], big software companies [Cor18] and academic research [DBL17].
The main idea behind the development of LLVM was to build a compiler that uses a static single assignment (SSA, explained in Section 2.1.1) compilation strategy and is independent of a particular programming language. This was achieved by creating the LLVM assembly language [llva], which is used as an intermediate representation (IR) for the program code inside the whole LLVM toolchain. Any high-level programming language can be compiled to this intermediate representation by a corresponding front-end (like Clang for C). All optimizations which are performed during compilation are done on this intermediate representation. Section 2.1.2 takes a closer look how the LLVM IR compares to assembly.

Figure 2.1.: LLVM logo [llvb]

Figure 2.2 shows the three major stages of a compilation using LLVM:

1. The frontend translates the input source code into the LLVM intermediate representation (IR) used inside the compiler.
2. The optimizer analyzes and transforms this intermediate representation. Most often this is done in order to achieve a better performance of the resulting program.
3. The backend generates the actual machine code for the target architecture.

While other compilers may treat these three steps as one big process (like illustrated in Figure 2.2 for GCC), they are much more independent from each other in the LLVM

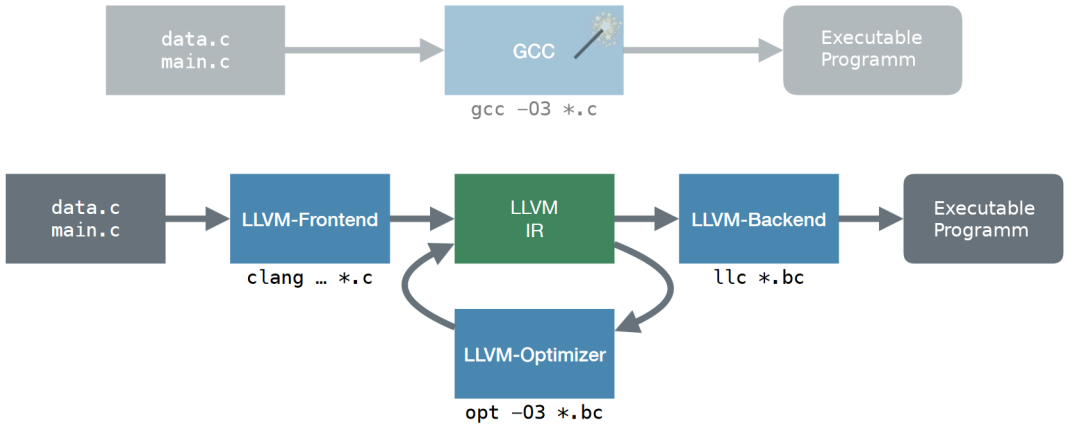

Figure 2.2.: Comparision of the major stages of a compilation process [Wit18b].

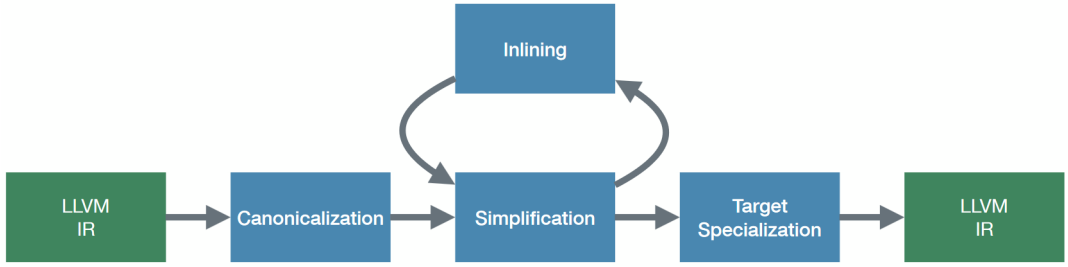

Figure 2.3.: LLVM pass pipeline [Wit18b].

compiler. The LLVM-backend produces machine code that is semantically equivalent to the output of every other compiler. The linking step of machine code is spared in Figure 2.2.

This modular approach implies that there can be different frontends for all kinds of programming languages and their only interaction with the optimizer is the IR code they generate. Therefore the optimizer is not language dependent. There are also different backends for different target architectures, that take IR code and translate it to actual machine code. This modular approach to optimizations is shown in Figure 2.3. The input is the IR, generated from the frontend. There are many passes for transformation and optimization [llvd]. Figure 2.3 shows a small excerpt of them as an example for optimization. The `opt` command also allows the user to run the optimization passes in any order. The output IR is than passed to the backend in order to generate the actual machine code.

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. In this thesis the in development LLVM version 8.0.0. [1] is used.

- 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 frontend 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. In this thesis thesis the in development Clang version 8.0.0. [2] is used.

- OpenMP: LLVM also provides its own implementation of the OpenMP standard to use with Clang. It fully supports the OpenMP 3.1 standard since the release of Clang 3.8.0. Additionally some OpenMP 4.0 and 4.5 features are already available [llve].

- 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.

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 transformation or optimization passes.

2.1.1. The LLVM intermediate representation

This section provides a brief overview of the LLVM intermediate representation. Further information about it can be found under `https://llvm.org/docs/LangRef.html`.

The LLVM intermediate representation (IR) is a low level, assembly-like language, that is used in the whole LLVM optimizer. Section 2.1.2 takes a closer look on how it compares to assembly.
There are three interchangeable forms of LLVM IR. First as an 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.

[1] `http://llvm.org/git/llvm.git` 051c6130853d67fc848d9b3b0b468e8c47b4b461
[2] `http://llvm.org/git/clang.git` 0b68110dff2427b00751de95404f437a8f934a33

```
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: Comparison between normal code and its SSA form [Ble17].

LLVM IR is a strongly typed, static single assignment (SSA) based programming language. The SSA form simplifies data flow analysis and many optimizations, which is why it is used by modern compilers. It originated as an algorithm to remove redundant computation [RWZ88].
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`.

In LLVM IR the name of each value starts with `%` or `@`. `%` means that it is a local value, which is only valid within the current function whereas `@` is used for global values. As functions are known on a global level, their name also starts with an `@`. This allows that there are no special keywords that can not be used as the name of a value. For example in line 15 of Listing 2.3 the result of the add operation is called add as well.

Note that the term value is used, as the assigned value to a "variable" may never change in SSA. Therefore LLVM does not know "variables". A variable is represented by a pointer to it and the value of the variable is influenced with load and store operations. The pointer to the variable is also a value, which may not change.

Instructions in LLVM IR are always inside of basic blocks. A basic block is a sequence of instructions with no branches, except for the end of the block. In LLVM IR each basic block has to be terminated with a terminator instruction e.g. a branch to another block or the return instruction of a function. Each block has a label that identifies it in order to use it as operands of a branch instruction. The first block of a function always has the label `entry`. As each basic block has to specify the next block(s) in its terminator instruction, the ordering of the basic blocks does not matter in LLVM IR.

One important instruction is the `alloca` instruction. `%ptr = alloca i32` yields a pointer to a new 32-bit integer variable. This variable is allocated at the stack, so it is only valid within the current function.
The `load` operation loads a value from a storage address.
For example `%val = load i32, i32* %ptr` loads the integer value from the pointer called `%ptr`. The `store` operation is analogous: `store i32 42, i32* %ptr` stores the value `42` to the location that `%ptr` points to.
This leads to the `getelementptr` instruction. The `getelementptr` instruction is used to calculate the address of an element of an array or struct.
`%arrayidx7.7.i = getelementptr inbounds double, double* %array, i64 %idx` returns the pointer to the element at index `%idx` of the array given through the pointer `array`.
As discussed above the `phi` instruction chooses one of the incoming values:
`%val = phi [ 1, %if.then ], [ %add, %if.else ]` chooses the value `1` if the execution flow came from block `%if.then`. There may be an arbitrary number of cases defined, one for each block that branches to this `phi` instruction. All the `phi` instructions have to be the first instructions of a block.
Additionally a `select` instruction exists: `%X = select i1 %cond, i8 17, i8 42` selects the value `17` if `%cond` is true. `42` would be selected if the condition is false. In general the `select` instruction allows to express basic if-conditions e.g. to find the maximum, without the need for IR-level branching. Therefore it may be viewed as the ternary operator (`? :`) of C/C++.

Many other operations such as `add`, `fadd` (f=float), `icmp`, `fcmp` for comparision or `call` to call a function are quite self-explanatory. One may see the next section for an example of LLVM IR code.

2.1.2. Comparing LLVM IR to assembler

In this section I will take a closer look on how the LLVM intermediate representation (IR) compares to Intel x86 assembler.

In Listing 2.2 a simple example C code is shown. Listing 2.3 shows the resulting LLVM IR, while Listing 2.4 shows the resulting x86 assembly code.

One similarity is that both codes are structured in basic blocks which have a label that can be branched to. The first block of a function is always named `entry` in LLVM IR. In contrast to LLVM IR, the x86 assembly does not contain the function signature (line 2 of Listing 2.3).

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

Listing 2.2: Example C Code.

```
1  ; Function Attrs: uwtable
2  define dso_local i32 @_Z3fibi(i32 %n) #0 {
3  entry:
4    %cmp = icmp sle i32 %n, 2
5    br i1 %cmp, label %if.then, label %if.else
6
7  if.then:                                          ; preds = %entry
8    br label %return
9
10 if.else:                                          ; preds = %entry
11   %sub = sub nsw i32 %n, 1
12   %call = call i32 @_Z3fibi(i32 %sub)
13   %sub1 = sub nsw i32 %n, 2
14   %call2 = call i32 @_Z3fibi(i32 %sub1)
15   %add = add nsw i32 %call, %call2
16   br label %return
17
18 return:                                          ; preds = %if.else,
     ↪ %if.then
19   %retval.0 = phi i32 [ 1, %if.then ], [ %add, %if.else ]
20   ret i32 %retval.0
21 }
```

Listing 2.3: LLVM IR of example code shown in Listing 2.2.

```
1   .globl  _Z3fibi                 # -- Begin function _Z3fibi
2   .p2align    4, 0x90
3   .type   _Z3fibi,@function
4 _Z3fibi:                              # @_Z3fibi
5   .cfi_startproc
6 # %bb.0:                              # %entry
7   push    rbp
8   .cfi_def_cfa_offset 16
9   .cfi_offset rbp, -16
10  mov rbp, rsp
11  .cfi_def_cfa_register rbp
12  sub rsp, 16
13  mov dword ptr [rbp - 8], edi
14  cmp dword ptr [rbp - 8], 2
15  jg  .LBB0_2
16 # %bb.1:                              # %if.then
17  mov dword ptr [rbp - 4], 1
18  jmp .LBB0_3
19 .LBB0_2:                              # %if.else
20  mov eax, dword ptr [rbp - 8]
21  sub eax, 1
22  mov edi, eax
23  call    _Z3fibi
24  mov edi, dword ptr [rbp - 8]
25  sub edi, 2
26  mov dword ptr [rbp - 12], eax # 4-byte Spill
27  call    _Z3fibi
28  mov edi, dword ptr [rbp - 12] # 4-byte Reload
29  add edi, eax
30  mov dword ptr [rbp - 4], edi
31 .LBB0_3:                              # %return
32  mov eax, dword ptr [rbp - 4]
33  add rsp, 16
34  pop rbp
35  .cfi_def_cfa rsp, 8
36  ret
37 .Lfunc_end0:
38  .size   _Z3fibi, .Lfunc_end0-_Z3fibi
39  .cfi_endproc
40                                      # -- End function
```

Listing 2.4: Intel x86 assembly of example code shown in Listing 2.2.

The LLVM IR abstracts from the calling convention of the particular system. This is needed for portability. In the x86 assembly one can see the setup of a new stackframe (lines 7 to 12 in Listing 2.4) and the wrap up (lines 33 to 36 in Listing 2.4). Also the way how the function gets the argument `%n` is abstracted away in LLVM IR to allow for different calling conventions. The argument of the function is refered as `dword ptr [rbp - 8]` in the x86 assembly[3].
One can also see that the LLVM IR abstracts away from the registers. In LLVM IR there are an infinite number of storage locations available. The LLVM backend will then map them to the actual used registers, whereas in the x86 assembly one can clearly see the load and store to the main memory. For example in line 20 of Listing 2.4 where the argument given to the function is loaded into the register `eax` in line 13 of Listing 2.4.
Another difference is the exact layout of the conditional branch (lines 4-5 in Listing 2.3 compared to lines 14-15 in Listing 2.4). In LLVM IR there is a compare instruction that determines if n is less or equal 2 [4] and the branch is based on the result of this comparison. This allows for much easier understanding of the branch condition compared to x86 assembly, where first n is compared against 2, which will also set the flag if it is greater than 2. The next instruction will branch if this flag was set, otherwise it just will continue with the next instruction. In LLVM IR every block has to be terminated with a terminator instruction (e.g. branch or return) where the next block that should be executed is given. This allows for the free reordering of blocks and easier trace of the control flow.

Overall the LLVM IR is easier to understand and manipulate than x86 assembly, while still strongly related to assembly code so that e.g. arithmetic optimizations can be done at a low level but without the need of knowing the machine specifics.

2.1.3. The LLVM API

This section provides a brief overview of the LLVM API. Further information about it can be found under ***`http://llvm.org/docs/ProgrammersManual.html`****. Also the documentation under* ***`https://llvm.org/doxygen/`*** *is quite helpful.*

As discussed above, one of the main features of LLVM is its modular structure, which allows for great expandability. As illustrated in Figure 2.3 each optimization is encapsulated in an LLVM pass, which analyzes or transforms the IR code - most often in order to give the program a better performance.

The execution of the individual passes is managed and scheduled by a pass manager. A pass manager can execute function and module passes. A function pass is run on every function of the module and has only access to a single function, while a module

[3] Actually the argument is given to the function in register `edi` and than stored to `dword ptr [rbp - 8]` in line 13 of Listing 2.3.

[4] Here a signed comparision is used. `sle` is the signed counterpart to `ule`, which is the same comparision for unsigned integers.

pass is run on the entire module, therefore it may modify anything inside that module. A module is one translation unit[5].

It is also possible to program your own passes. They have to be derived from the module or function pass base class. The base class provides a virtual `runOnModule` (or `runOnFunction`) function, that is invoked by the pass manager to execute the pass. This function takes the module (or function) as an argument and is expected to return `true`, if the pass has done any modifications to the IR, `false` otherwise. Passes that do not alter the IR are considered analysis passes. The analysis results may be accessed by other passes. The LLVM API offers the `GetAnalysisResult` method for this purpose.

In order to load a self developed pass, Clang offers the `-Xclang` option, that is used to pass options to the compiler backend. Therefore one may execute a pass (given by the shared opject file `pass.so`) using `-Xclang -load -Xclang pass.so` as compiler flags during compilation time.

Passes are written in C++. The pass implemented in this thesis is a transformative module pass, as it will alter the whole module (see Chapter 5 for a description of the developed pass). The next subsections describe some of the important classes the LLVM API offers to manipulate the IR.

Module Class

"Modules are the top level container of all other LLVM Intermediate Representation (IR) objects. Each module directly contains a list of globals variables, a list of functions, a list of libraries (or other modules) this module depends on, a symbol table, and various data about the target's characteristics."[LLV18a]
The Module class offers many methods to access this globally defined objects (e.g. `getIdentifiedStructTypes` to get a list of all struct types). It also offers methods to insert new global variables. For example the `getOrInsertFunction` method looks for a function with the given name and type and will return it [6] or insert a new empty function if no function was found.

Function Class

The Function class represents a function, therefore it contains a list of the BasicBlocks the function consists of. It also allows access to the function arguments, as well as the metadata associated with this function. A function is also a (global) value, as it may be used as one operand of a call instruction.

[5] "translation unit" according to ISO/IEC 9899 [IC07].
[6] Casted to the given function type if necessary.

Value Class

The Value class is one of the most important classes. As discussed before, a value in LLVM IR is assigned once and may never change (SSA). Therefore everything can be considered a value in LLVM IR[7]. This means that Value is basically the super class to all other important LLVM classes, such as instructions. A value allows access to its users, respectively all instructions where the value is used. This allows for an easy traversion of the control flow.

Instruction Class

Instruction is the base class for all instructions of LLVM IR. Instruction is a subclass of Value, as the result of each instruction is a value[8]. In order to cast an Instruction to the specific subclass (e.g. `CallInst` for a call instruction) LLVM offers the `dyn_cast` function, which will return a nullpointer when the cast operation is not allowed.
Therefore the `dyn_cast` is often used in an if-statement, e.g.
`if (auto *call = dyn_cast<CallInst>(Val))` which branches into the if, only if `Val` is a call instruction. Every type of instruction has its own subclass, which offers methods to access or manipulate the specific instruction. E.g. the `CallInst` offers a `setCalledFunction` method to change the callee of this call instruction.

Type Class

The Type class represents the type of a value. "Only one instance of a particular type is ever created. Thus seeing if two types are equal is a matter of doing a trivial pointer comparison." [LLV18b]. The Type class offers methods like `isDoubleTy` to assess the specific type.

IRBuilder Class

The IRBuilder class is the main way how one can create and insert new instructions. It offers many methods to create new instructions like `CreateAdd`. The IRBuilder will need an insertion point for insert new instructions. New instructions will be inserted before the insertion point given by the `SetInsertPoint` method or the constructor of the IRBuilder class. Most often an insertion point is an already present instruction or the end of a (often newly created) basic block.

2.1.4. Clang API

This section (based on [cla18b]) describes the Clang API used to define and handle new pragma directives.

Listing 2.5 shows the usage of the API to handle the `#pragma example_pragma` directive.

[7] Except the module itself and types (of values).
[8] Also Store instructions which have no "result value" are values in LLVM.

```
1  // Define a pragma handler for #pragma example_pragma
2  class ExamplePragmaHandler : public PragmaHandler {
3  public:
4    ExamplePragmaHandler() : PragmaHandler("example_pragma") { }
5    void HandlePragma(Preprocessor &PP, PragmaIntroducerKind
       ↪ Introducer, Token &PragmaTok) {
6      /* handle the pragma */
7    }
8  };
9
10 static PragmaHandlerRegistry::Add<ExamplePragmaHandler>
     ↪ Y("example_pragma","example pragma description");
```

Listing 2.5: Example of a pragma handling Clang plugin [cla18b].

The `HandlePragma` function is run whenever a `#pragma example_pragma` directive is encountered by the preprocessor.
The preprocessor object is given as the first argument to the `HandlePragma` function. The preprocessor object can be used to analyze and manipulate the processed source. For example one can change the following tokens or insert new ones in order to alter the following statement. The second argument gives the type of the token that introduced this pragma directive[9]. The third argument is the pragma token, that was used to decide which pragma handler will be invoked (here it is the `example_pragma` identifier token).

At this stage of compilation, the preprocessor is tokenizing the source code in order to feed it to the parser [cla18a]. A token is a single element of a programming language like C++. There are different kinds of tokens, for example keywords such as `return` or `if`, seperators such as `,` or `;` or literals such as `"example string"` or `42`. The Token class combines all the information for a token, such as the type of the token, its position in the source code and the associated value if it is a literal token.

The Clang API also provides `PragmaNamespace`, which are derived from `PragmaHandler` and "subdivides the namespace of pragmas, allowing hierarchical pragmas to be defined" [cla18c], such as `#pragma omp`. Therefore the `void PragmaNamespace::AddPragma( PragmaHandler * Handler)` method exists, in order to register new pragmas for a pragma namespace.

Clang plugins that define new pragmas can be loaded in the same way as custom LLVM passes using the `-Xclang -load -Xclang plugin.so` command line options.

[9] Pragmas may be started with `#pragma example`, `_Pragma("example")` or `__pragma(example)`.

2.2. MPI

The Message Passing Interface (MPI) has become the lingua franca of parallel programming a distributed memory system [Lud17].

The MPI standard defines many routines for managing communication and synchronization of a distributed parallel application. A MPI program must be initialized with the `MPI_Init` routine and should only exit after executing the `MPI_Finalize` routine. Each process gets assigned a unique number -called rank- that is in the range of 0 to the number of processes available, so that the processes are numbered. Processes are part of one or more communicators. Communicators define groups of processes that can communicate with each other using MPI communication routines. Every process is part of the predefined communicator `MPI_COMM_WORLD` but MPI also provides routines like `MPI_Comm_split` or `MPI_Comm_create` in order to create new communicators.

The next subsections take a closer look on the different kind of communication routines.

2.2.1. MPI point to point communication

This section (based on [Bar14b]) provides a brief overview of the concept of MPI point to point communication. Further information about it can be found under ***https://cvw.cac.cornell.edu/MPIP2P/****.*

Point to point is the most commonly used communication pattern in MPI [Bar14b]. It involves the transfer of a message from one specific process to another specific process in the communicator.

The sending process must call some form of `MPI_Send` in order to send a message to another process.

```
1 int MPI_Send(
2     const void *buf,      //initial address of send buffer (choice)
3     int count,            //number of elements in send buffer
        ↪ (nonnegative integer)
4     MPI_Datatype datatype,//datatype of each send buffer element
        ↪ (handle)
5     int dest,             //rank of destination (integer)
6     int tag,              //message tag (integer)
7     MPI_Comm comm)        //communicator (handle)
```

Listing 2.6: Signature of the `MPI_Send` Function [For15].

The signature of the `MPI_Send` function is shown in Listing 2.6. The calling process sends a message from the send buffer consisting of `count` elements of the specified `datatype` to the destination process (rank). Each message has a message tag in order to distinguish different messages. The target process must call a matching `MPI_Recv` function. The

signature is basically the same as for `MPI_Send` [10]. Instead of `dest` the parameter is called `source` and refers to the rank of the sending process, that called `MPI_Send`. One can also use the predefined value `MPI_ANY_SOURCE` to look for a message with the given message tag from any other processor. Similarly another predefined value `MPI_ANY_TAG` exists, to look for a message with any message tag[11]. MPI offers several modes of point to point communication:

- (standard) **Blocking** (`MPI_Send`): In the standard mode the call to `MPI_Send` blocks until the sending buffer is ready to be used again. This does not necessarily mean that the receiver has already received the message. The exact behaviour is implementation specific.
- **Nonblocking** (`MPI_Isend`): A nonblocking call immediately returns. The message is sent in the background when the receiver is ready to receive it (has called `MPI_Recv`). Therefore the sending buffer must not be overwritten. It is necessary to call `MPI_Wait` in order to wait for the `MPI_Isend` to complete, before reusing the message buffer.
  - Also a nonblocking counterpart `MPI_Irecv` exists. This call is similar to the nonblocking `MPI_Isend`. It immediately returns but the receive buffer may not be used until `MPI_Wait` was used in order to wait for the full transmission of the message.
- **Buffered** (`MPI_Bsend`): A buffered send operation copies the message to a buffer and then returns. The message is buffered until the receiver is ready to receive the message (has called `MPI_Recv`). Only then the message is transmitted to the receiver and the buffer space is again free to use with subsequent calls of `MPI_Bsend`. Therefore users have to supply enough buffer space with the `MPI_Buffer_attach` function in order to work properly.
- **Synchronous** (`MPI_Ssend`): A syncronous message transfer only starts if the receiver is ready to receive the message (has called `MPI_Recv`). Therefore the call to `MPI_Ssend` will block in order to wait for the receiver to be ready.
- **Ready** (`MPI_Rsend`): This sending operation only blocks long enough to send the data to the network. However if the matching receive has not already been posted when the send begins, an error will be generated.

[10] `MPI_Recv` has an additional parameter: `MPI_Status *status` which will initialize a status object which contains information about the status of the received message such as how many bytes were received. One can also use the predefined `MPI_STATUS_IGNORE` value in order to ignore the status.

[11] The message tag and sending process of the received message can be found out with the help of the status object.

2.2.2. MPI collective communication

This section (based on [Bar14a]) provides a brief overview of the concept of MPI collective communication. Further information about it can be found under ***https://cvw.cac.cornell.edu/MPIcc/****.*

All of the processes in a communicator are involved in a collective communication operation. One can view collective communication operations as a substitute for a more complex series of point to point communication operations. But the usage of collective communication should generally be preferred if applicable, because they allow an optimized implementation [Lud17, p. 147].

In general there are three types of collective operations:

- **Synchronization**
  - `MPI_Barrier`: A barrier is a simple synchronization mechanism: any process calling it will be blocked until all the processes within the communicator have called it.
- **Data movement**
  - `MPI_Bcast`: A broadcast can be used if one root process needs to send the same data to all other processes.
  - Other forms of data movement like `MPI_Scatter` and `MPI_Gather` can be used when the root process wants to distribute the data among all processes so that each process gets a distinct chunk of the data.
- **Global computation**
  - `MPI_Reduce`: A global reduction is performed in order to combine the data from the different processes (e.g. sum it up). Many basic reduction operations (like sum or min) are predefined. It is also possible for the user to define their own reduction operation.
  - `MPI_Scan` may be used to perform partial reductions.

The MPI-3 standard also introduced non blocking collective communication [For15] like `MPI_Ibcast`. They work in an analogous way as non blocking point to point communication routines work (see Section 2.2.1).

2.2.3. MPI one sided communication

This section (based on [LB14]) provides a brief overview of the concept of MPI one sided communication. Further information about it can be found under ***https://cvw.cac.cornell.edu/MPIoneSided/****.*

One sided communication methods were added to MPI as a part of the MPI-2 standard

and expanded in the MPI-3 version [For15]. One sided communication was designed to further exploit the Remote Memory Access (RMA) functionality, which is provided by low-latency interconnect hardware such as InfiniBand.

For MPI one sided communication each process may expose certain local memory to the usage of one sided communication methods. This is done through a MPI window, which is created with the `MPI_Win_create` function shown in Listing 2.7.

```
1 int MPI_Win_create(
2      void *base,   //initial address of window (choice)
3      MPI_Aint size, //size of window in bytes (nonnegative integer)
4      int disp_unit, //local unit size for displacements, in bytes
          ↪ (positive integer)
5      MPI_Info info, //info argument (handle)
6      MPI_Comm comm, //communicator (handle)
7      MPI_Win *win)  //window object returned by the call (handle)
```

Listing 2.7: Signature of the `MPI_Win_create` function [For15].

`MPI_Win_create` is a collective operation. The info argument provides optimization hints to the runtime about the expected usage pattern of the window. One can use `MPI_INFO_NULL` as a default value. Each process can expose a different amount of memory to the window, including 0 bytes. Once the window is created, it can be used by all processes in the communicator to access the exposed memory with one sided communication calls like `MPI_Put` or `MPI_Get`. The signature of `MPI_Put` is shown in Listing 2.8.

```
1 int MPI_Put(
2     const void *origin_addr,      //initial address of origin buffer
3     int origin_count,             //number of entries in origin buffer
4     MPI_Datatype origin_datatype,//datatype of each entry in origin
          ↪ buffer
5     int target_rank,              //rank of target
6     MPI_Aint target_disp,         //displacement from start of window
          ↪ to target buffer
7     int target_count,             //number of entries in target buffer
8     MPI_Datatype target_datatype,//datatype of each entry in target
          ↪ buffer
9     MPI_Win win)                  //window object used for
          ↪ communication
```

Listing 2.8: Signature of the `MPI_Put` function [For15].

The `MPI_Put` method writes the data in the origin buffer to the specified position (`target_disp`) within the other processes (`rank`) exposed memory location, which was chosen at the window creation with `MPI_Win_create`.

The target rank does not have to call a matching "receive" operation, as the communication is one sided. Nevertheless the one sided communications need to be synchronized with exposure epochs, in which the communication may occur. There are several ways for synchronization of one sided communication in MPI:

- **Fence synchronization**: Both, the start and the end of an RMA epoch, is defined by all processes calling the collective `MPI_Win_fence` call. RMA communication calls cannot begin until the target process has called `MPI_Win_fence`. When `MPI_Win_fence` is called to terminate an epoch, the call will block until all outstanding RMA operations are completed.

- **Post-Start-Complete-Wait**: This method of synchronization is similar to non-blocking point to point communication. The origin process must call `MPI_Win_start` to start an epoch and `MPI_Win_complete` to complete it. The target process must call `MPI_Win_post` to start the exposure epoch and `MPI_Win_wait` to end it. `MPI_Win_wait` will block until all outstanding RMA operations have finished.

- **Lock-Unlock**: Locking is a passive target synchronization method. The origin process must call `MPI_Win_lock` to start the exposure epoch and `MPI_Win_unlock` to end it. The target process does not need to call any methods for synchronization. A window cannot be exposed via another exposure epoch (e.g. by fence synchronization) and be locked concurrently.

MPI also supports the creation of dynamic windows with the `MPI_Win_create_dynamic` function. A process can dynamically add more memory to a dynamic window for exposure to the other processes with the `MPI_Win_attach` function. The process later can revoke the exposure of an attached memory region by calling `MPI_Win_detach`. In order to calculate the displacement for such dynamic windows MPI provides the `MPI_Get_address` function.

2.3. OpenMP

This section (based on [Bar17]) provides a brief overview of OpenMP. Further information about it can be found under ***https://computing.llnl.gov/tutorials/openMP/****.*

"OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs." [Boa17]

In order to achieve parallelism of a block of code, OpenMP defines compiler directives like `#pragma omp parallel` . The block of code following the `omp parallel` pragma directive will then be executed by as many threads as CPU cores are available by default. The number of threads used to execute a parallel region can be influenced by setting the `OMP_NUM_THREADS` environment variable, using the OpenMP provided function `omp_set_num_threads` or specifying a `num_threads` clause within the parallel directive.

As a parallel region is executed by different threads, all memory is shared between them by default. One may specify the `default` clause to change this behaviour. I suggest

to use `default(none)` so that it must be stated explicitly whether or not a variable is shared, which is helpful when understanding and debugging OpenMP code. One may specify the `private` clause in order to tell OpenMP that each thread should use a local version of the variable. A private variable is undefined at the beginning of a parallel section. If one wants to initialize the variable with the value it has before the parallel region, one may use the `firstprivate` clause. The `shared` clause indicates that a variable should be shared among the threads.

In order to divide the computation between the different threads, OpenMP offers several worksharing directives like the loop construct `omp for`. The `omp for` directive splits up the iterations of the following for loop. Each thread gets assigned a subset of the total loop iterations. Therefore it has to be ensured by the programmer that no data dependency between two loop iterations exist, as the ordering of loop iterations are not defined[12]. The division of the loop iterations among the threads may be influenced using the `schedule` clause. An example of the usage of the OpenMP clauses is provided in Listing 2.9.

```
1 #pragma omp parallel for shared(a) schedule(static) reduction(+:b)
2 for(int i = 0; i < 100; ++i){// loop index i is implicit private
3     int priv = 0;// as priv is only defined within the scope of this
          ↪ parallel region, it is private
4     /* do some work */
5 }
```

Listing 2.9: Example of the usage of the OpenMP clauses.

As shown above, the `omp parallel for` is the combination of the `omp parallel` and `omp for` directives and indicates a parallel region whose only content is the shown loop construct.

The Listing 2.9 also shows the usage of the reduction clause. With this clause the variable (`b`) is private[13] but at the end of the parallel region the local values are combined using the given operator (`+`). OpenMP supports many basic reduction functions like `+,*,min,max`. Newer versions of OpenMP (from version 4.0) also allow user defined reductions [Ope13]. User defined reductions are declared using the `omp declare reduction` pragma.

In order to avoid race conditions and synchronize the threads, OpenMP provides several runtime methods such as `omp_set_lock` as well as the `omp barrier`, `omp critical` or `omp atomic` directives. The barrier marks a synchronization point at which a thread has to wait until all other threads also have arrived. The critical pragma marks a region which must be executed by only one thread at a time. The atomic pragma indicates that the next instruction should be performed atomically, which has basically the same

[12] Unless an `omp ordered` directive is used within the loop.

[13] And initialized with 0 or 1 depending on the reduction operator.

semantics as a critical directive[14].

OpenMP uses a fork-join concept. This means that at the beginn of a parallel region all (new) threads are started (fork) while at the end of a parallel region, once all threads are completed[15], the execution of the following sequential part is only continued by the main thread (join).

Another possibility to spread the work among several threads is the `omp task` pragma. The task directive defines an explicit task that may be executed by any thread. Tasks may only be used within a parallel region. Therefore the creation of the tasks is usually within a `omp single` region, that is executed by only one thread. Any thread within a parallel region may start working on the new task. `omp taskwait` is used to wait for the completion of all outstanding tasks. A `omp barrier` implies a taskwait and before the join of all threads at the end of a parallel region, an implicit taskwait is also performed.

If an OpenMP program is compiled without the compiler flag `-fopenmp` set - or with a compiler that does not support OpenMP - the compiler directives will be ignored and the program will compile as a normal sequential program. This also makes it easy to use OpenMP to parallelize existing sequential programs.

[14] Atomic is more lightweight but only allows certain operations like increment.
[15] So there is an implicit barrier.

3. Overview of the considered use cases

Within this chapter I will give an overview over the different use cases considered in my thesis.

Phil Colella identified seven numerical methods which will be important for science and engineering for at least the next decade in his opinion [Col04]. They become known as the seven dwarfs. Based on his work, the widely read "Berkeley View" report [ABC+06] added another six "dwarfs".

The dwarfs create classes of programs, where each class is defined by similarity in computation and data movement. They are specified at a high level of abstraction in order to allow to investigate their general behaviour rather than investigating one particular program. The programs that are members of a particular class may be implemented differently or evolve over time, but the claim is that the underlying patterns have persisted through generations of changes and will remain important into the future [ABC+06].

Here I will give a brief overview over the seven "original" [Col04] and the six new dwarfs found in the "Berkeley View" report [ABC+06], while the remaining subsections describe the dwarfs more in detail:

1. **Dense Linear Algebra** (Section 3.1): Data are dense matrices or vectors. Generally, such applications use unit-stride memory accesses to read data from rows, and strided accesses to read data from columns.

2. **Sparse Linear Algebra** (Section 3.2): Data sets include many zero values. Data is usually stored in compressed matrices to reduce the storage and bandwith requirements to access all of the nonzero values.

3. **Spectral Methods** (Section 3.3): Data are in the frequency domain, as opposed to time or spatial domains. Typically, spectral methods use multiple butterfly stages, which combine multiply-add operations and a specific pattern of data permutation, with all-to-all communication for some stages and strictly local for others.

4. **N-Body Methods** (Section 3.4): Depends on interactions between many discrete points. Variations include particle-particle methods, where every point depends on all others.

5. **Structured Grids** (Section 3.5): Represented by a regular grid; points on grid are conceptually updated together. It has high spatial locality. Updates may be in place or between two versions of the grid.

6. **Unstructured Grids** (Section 3.6): An irregular grid where data locations are selected, usually by underlying characteristics of the application. Data point location and connectivity of neighbouring points must be explicit. The points on the grid are conceptually updated together. Updates typically involve multiple levels of memory reference indirection, as an update to any point requires first determining a list of neighbouring points, and then loading values from those neighbouring points.

7. **Monte Carlo** (Section 3.7): Calculations depend on statistical results of repeated random trials. Considered embarrassingly parallel.

8. **Combinational Logic (e.g., encryption)** (Section 3.8): Generally involves performing simple operations on very large amounts of data often exploiting bit-level parallelism.

9. **Graph Traversal (e.g., Quicksort)** (Section 3.9): Applications must traverse a number of objects and examine characteristics of those objects such as would be used for search.

10. **Dynamic Programming** (Section 3.10): An algorithmic technique that computes solutions by solving simpler overlapping subproblems.

11. **Backtrack and Branch+Bound** (Section 3.11): These involves solving various search and global optimization problems for intractably large spaces.

12. **Construct Graphical Models** (Section 3.12): Applications involves graphs that represent random variables as nodes and conditional dependencies as edges. Examples include Bayesian networks and hidden Markov models.

13. **Finite State Machine** (Section 3.13): A finite state machine represents an interconnected set of states, such as would be used for parsing.

One advantage of using these higher-level descriptions of programs is that I am not tied to code that may have been written originally to run optimal on another system [WWP09]. According to the motivation of the thesis that OpenMP is a simple tool of parallelizing a program, I will discuss only one specific way to parallelize the particular use cases with OpenMP. The way chosen should represent a basic version of parallelizing the respective use case with OpenMP.
Of course many optimizations (particulary with regards to load balancing) can be made to the presented parallelizations. This thesis only covers static schedulings, as scheduling is an exhaustive topic itself, scheduling is mostly spared out for this thesis and left for future work (Section 8.2). But all this optimizations require a deep understanding of parallel processing in general as well as the particular problem, the used technology and

the machine the application should perform on. As the goal of this thesis is to determine whether automatically translating OpenMP programs to MPI using LLVM, in order to enable people, that do not have all this special insights (and do not need to know MPI), to scale up their applications beyond the limits of OpenMP, it seems not feasible to use highly optimized parallel programs to evaluate the possibilities. As the intelligent authors of such highly optimized programs surely have the capabilities to effortlessly learn and use MPI if this is necessary.

As the claim of the presented "dwarfs" is that all computation belonging to one dwarf is the same *type* of computation, it is feasible to use this high level descriptions of the computation, that should be performed, in order to assess if this *type* of computation in general can benefit from automatically translating it from OpenMP to MPI using LLVM.

The remainder of this chapter describes the dwarfs more in detail and also shows the specific algorithms used to demonstrate the automatic translation with LLVM.

3.1. Dense Linear Algebra

The field of linear algebra is a fundamental branch of mathematics. It is concerned with linear equations, matrices and vector spaces. Linear algebra is a fundamental basis for many fields of science [And14a]. The very broad range of applications for linear algebra ranges from optimizing production processes with the simplex-algorithm [And14b, Dan63] to satellite positioning navigation such as GPS [PEASJ96].

In computer science linear algebra problems are often partitioned into dense linear algebra and sparse linear algebra problems. Although there is no strict boundary between this two classes of linear algebra, problems where a majority of the elements in a vector or matrix are zero are considered sparse [Tim16]. Nevertheless this classification is relevant for the design of efficient data structures and algorithms, but does not originate from the underlying mathematical concepts.

3.1.1. Example: Solving linear equation systems

One of the most basic procedures in linear algebra is to solve a linear equation system. The most famous algorithm to perform this is the Gaussian elemination method [And14a].

One way to parallelize this method is to assign each threads some rows of the matrix to work on. In each elimination step the current line has to be shared with all other threads, so that all threads can perform the calculation on their assigned rows independently. This is illustrated in Figure 3.1.

A more detailed explanation of the algorithm and the usage of OpenMP within this case with the help of a listing can be found in Appendix A.1.

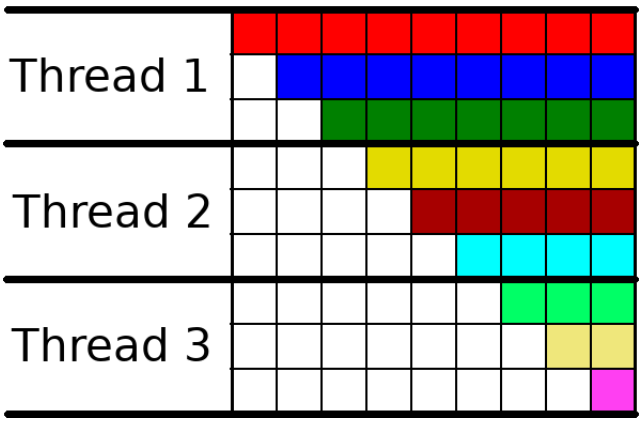

Figure 3.1.: Distributing data for solving a system of linear equations among three threads. The colors indicate which data has to be shared among all threads in each elimination step.

3.2. Sparse Linear Algebra

As this thesis can only cover a subset of the different cases, I choose to spare out this case, as it may be viewed as a special subset of the dense linear algebra case covered in Section 3.1, where much of the encountered values are zero.

3.3. Spectral Methods

In spectral methods data is in the frequency domain and not in a (discrete) time or spatial domain.

Spectral methods are also used to solve differential equations. A spectral method may be seen as a very special case of a method of weighted residual. Method of weighted residuals use trial functions to approximate the solution of a differential equation. In spectral methods the trial functions are infinitely differentiable global functions. In contrast, finite-element of finite-difference methods use trial functions, that are local to one datapoint (compare Section 3.5) [CHQ+12]. Spectral methods have good error properties, when the solution is smooth, though Gibbs phenomenon might lead to higher errors when approximating a discontinuous function [HH79].

Spectral methods often use a specific pattern of data permutation. Some stages require all-to-all communication and other stages might be strictly local operations [ABC+06].

3.3.1. Example: Fourier transform

One of the most important technique in spectral methods is a Fourier transform [CT65]. It has a very broad range of applications from compression of audio data [Bos04] to option pricing [CM99]. Figure 3.2 illustrates the main goal of a Fourier transform: to

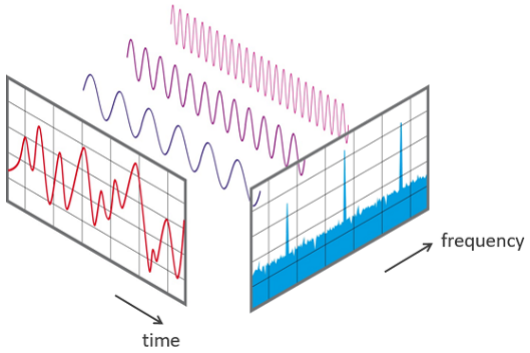

Figure 3.2.: View of a signal in the time and frequency domain (from [Wika]).

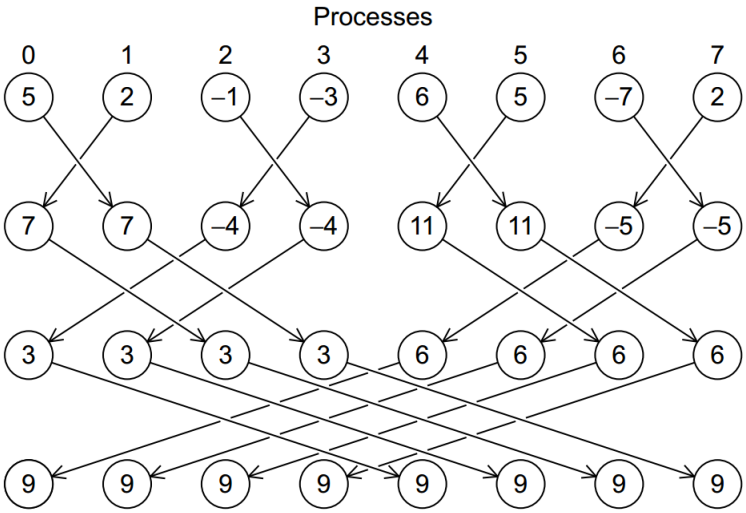

Figure 3.3.: Visualisation of a butterfly communicatin scheme. Here a sumation operation is visualized(from [Fin]).

transform a signal from the time to the frequency domain. For example to filter out the noise of an audio signal. The transformation may also be done in reverse to transform a signal (back) to its time domain.

The Cooley–Tukey FFT algorithm [CT65] in its in-place version consists of two phases: first a bit-reversal of the data indices (a reordering of the data), second the application of the Danielson–Lanczos lemma [DL42], which will result in a "butterfly" communication scheme illustrated in Figure 3.3.

A more detailed explanation of the algorithm and the usage of OpenMP within this case with the help of a listing can be found in Appendix A.2.

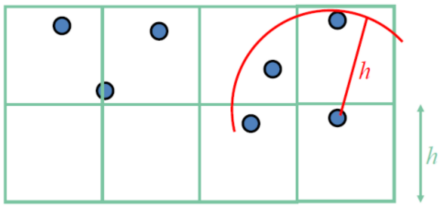

Figure 3.4.: Illustration of the cutoff radius [yK09].

3.4. N-Body Methods

In N-Body methods every datapoint depends on many other points. A good example for this is a particle simulation where one particle interacts with all other particles.
Other forms of particle simulations make use of a cutoff radius in order to increase the simulation's performance, because it is not necessary to compute the interaction of particles that are not close together as these interactions will not be significant anyway [CJ12]. This form can be viewed as a structured grid model. The particles are sorted into a grid where only the interactions between particles in neighbouring grid cells have to be evaluated. This is illustrated in Figure 3.4, where only the interactions between particles in neighbouring (or same) cells need to be evaluated as all other particles have a distance greater than the cutoff radius.
Therefore I will spare out a deeper look at N-Body methods in this thesis and rather focus on structured grids (Section 3.5). Nevertheless it will be an interesting topic for future work (Section 8.2) if also methods where one point depends on all other could benefit from the increased scalability of MPI compared to OpenMP.

3.5. Structured Grids

Key feature of this class is, that the data is stored in a regular grid. Points of the grid are conceptually updated together. Therefore each thread often operates on "his own" chunk of the data. Only at the boundary of two chunks information needs to be shared. A prominent example for this is the calculation of a partial differential equation. Figure 3.5 illustrates this data distribution for a two dimensional calculation. This type of computation, where each datapoint depends only on itself and the neighbouring points is called a stencil code [RMcKB97].
A distribution of a three dimensional room can be done analogous. The boundary region that needs to be shared with the neighbouring thread then is a plane of the cube instead of a line of the matrix.

Also some tasks of image analysis, like calculating gradient images for edge detection, have the same memory access pattern [Can87].

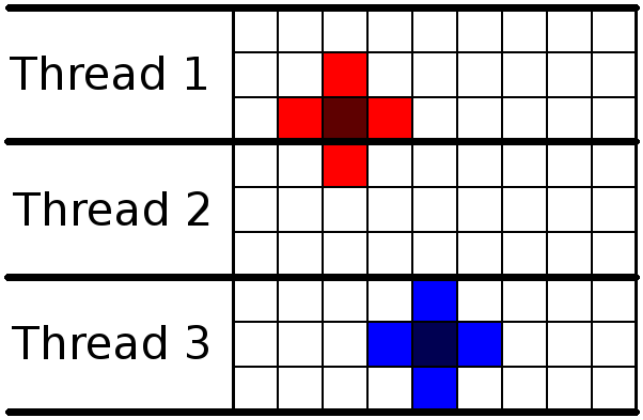

Figure 3.5.: Data distribution when solving a partial differential equation. Red indicates a situation where data needs to be shared among two threads. Blue indicates a situation where a thread only uses local data and does not interfere with other threads.

3.5.1. Example: Partial differential equation

One example for this is solving a partial differential equation (PDE) with a stencil code (as illustrated in Figure 3.5), for example to model heat diffusion. I will not use an inplace variant, as less synchronization is needed, if there are no data-dependencies between the threads.

A more detailed explanation of the algorithm and the usage of OpenMP within this case with the help of a listing can be found in Appendix A.3.

3.6. Unstructured Grids

Programs that are considered part of the unstructured grid class, share many key features with the structured grid class (Section 3.5). Also the data is stored in a "grid", where the points on the grid are conceptually updated together. The key difference is that the "grid" is not regular. Therefore the neighbour relation of the grid points has to be made explicit. This typically leads to multiple levels of memory reference, as an update to a grid point first requires to look up the neighbouring points and then load their values in order to compute the updated value.
As the "structure" of the grid is determined by the specific application, I will use the similar but more consistent structured grids class for further investigation within my thesis. Nevertheless this thesis is also including a *proof of concept* that it is possible to translate linked data structures with many levels of memory reference to MPI (see Section 6.11).

3.7. Monte Carlo

Monte Carlo methods where a series of independent random trials is done are *embarrassingly parallel* [ABC+06]. This means, that there is no data dependency between the threads. This class of applications is also often referred as map-reduce tasks. Map-reduce tasks consist of two phases: a map and a reduce phase. They are commonly used in big data analysis [Kun16].

While the map part of a map-reduce task can be computed independently once the data is distributed among the threads (*embarrassingly parallel*), the reduce part might be computed serially in OpenMP. The OpenMP standard does not specify if the reduction clause has to be computed concurrently [Ope13]. Therefore some implementations of OpenMP execute the reduction serially [lib] in order to avoid cache misses and synchronization overhead. But for complex reductions it is very important that they are computed as parallel as possible.

Note that Monte Carlo experiments also have a reduce phase where the different results from the independent random trials are aggregated. Analogous the map phase is the phase, in which the different random trials have been done independently, although there might be no data transformation as in the context of a map-reduce task.

3.7.1. Example K-Means Clustering

As an example of a map-reduce task I will use K-Means clustering.
In the map phase the datapoints are mapped to the nearest cluster centers. In the next phase all datapoints of a cluster need to be "reduced" to the new more optimal cluster centers. This process is repeated until nearly no more datapoints changes the cluster (Lloyd's algorithm [TM16, Llo82]). One can start with random initial cluster centers.

A more detailed explanation of the algorithm and the usage of OpenMP within this case with the help of a listing can be found in Appendix A.4.

3.8. Combinational Logic

"Combinational Logic generally involves performing simple operations on large amounts of data often exploiting bit-level parallelism. For example, computing Cyclic Redundancy Codes (CRC) is critical to ensure integrity."[ABC+06] As the focus of this class of computation is the exploitation of bit-level parallelism, I will not include it in this thesis, as MPI is a higher level approach.

3.9. Graph Traversal

A graph traversal application visits many nodes in a graph by following successive edges. Therefore these applications typically involve many levels of indirection and a relatively small amount of computation.
The many levels of indirection might be very costly when traversed by an MPI application, especially when the graph is irregular, as the graph may be distributed among many nodes and one traversal therefore might need to communicate with many nodes.
Another common feature is, that it is often a good idea to spawn new threads if needed. For example whenever a visited node has more than one outgoing edge the application can spawn a seperate thread to follow each of the edges.
While this is possible in MPI (e.g. using the `MPI_Comm_spawn` function) [mpi], the idea of spawning new processes during the runtime, if this is beneficial to the performance, is contradictory to the idea that I will only use static scheduling strategies within this thesis. Additionally one of the main differences between a process and a thread (or sometimes lightweight process) is that threads are more lightweight to spawn and destroy, with not much memory copying involved [Vah96].
Even though this is an interesting topic for future work (Section 8.2), I will not take a closer look on this type of computation within this thesis.
Nevertheless this thesis is also including a *proof of concept* that it is possible to translate linked data structures with many levels of memory reference - such as graphs - to MPI (see Section 6.11).

3.10. Dynamic Programming

One popular technique for solving optimization problems is dynamic programming [KT06].

Dynamic programming can be applied, if a problem can be solved by recursively splitting it into smaller sub problems and then finding the optimal solution for the smaller sub problems (optimal substructure). In order to avoid recomputing of the sub problems, the best solution for a sub problem is stored in a matrix. The matrix then is computed iteratively. This means that first the smaller problems are computed and then the larger ones may use the already computed solution for the smaller sub problems.
Often the matrix only stores the score of the solution. If the actual solution is needed it can be gained by backtracking through the computed sub problems (see Section 3.10.1 for an example).

While the final step of backtracking the best solution is sequential most times, one may exploit parallelism when computing the matrix with the scores of the sub problems. Figure 3.6 shows how the computation of the matrix with the solution scores can be blocked. Blocks of the same color may be computed in parallel. Once the best possible score of the smallest sub problems in the red block at the top left have been computed, the computation on the two blue blocks is independent of each other.

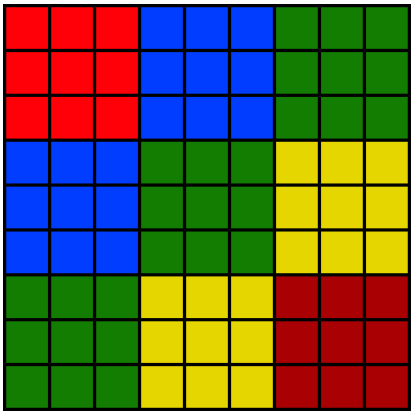

Figure 3.6.: Grouping the computation of the solution score matrix. The blocks of the same color may be computed in parallel as they are independent from each other.

3.10.1. Example: Smith-Waterman Algorithm

As an example for this technique I will use the Smith-Waterman algorithm for sequence alignment [SW81]. For example this algorithm is used in the field of bioinformatics to find regions of similarity in a sequence of DNA, RNA or protein, in order to discover functional, structural or evolutionary relationships [Rom04].

In the Smith-Waterman algorithm the alignment score is calculated with the following recursive Equation (3.1):

$$H_{ij} = max \begin{cases} H_{i-1,j-1} + s(a_i, b_j), \\ max_{k \geq 1}\{H_{i-k,j} - W_k\} \\ max_{l \geq 1}\{H_{i,j-l} - W_l\} \\ 0 \end{cases} \tag{3.1}$$

where H_{ij} is the alignment score of the sequences $a_1, a_2, ..., a_i$ and $b_1, b_2, ..., b_j$,
$s(a_i, b_j)$ denotes the similarity of the symbols a_i and b_j
and W_k is the cost of having a gap with length k in the alignment.
The score 0 is set if no alignment was found so far. H_{0j} and H_{i0} are also initialized with 0.

The usage of Equation (3.1) is visualized in Figure 3.7[1]. Left the case where no alignment was found so far is illustrated. On the right side, the symbols match. Therefore the highest score is the match of theese two symbols. The -2 arrows from above and right illustrate case 2 and 3 for Equation (3.1), where a gap in the alignment exists.

A more detailed explanation of the algorithm and the usage of OpenMP within this case with the help of a listing can be found in Appendix A.5.

[1] A good visualisation of a full example including the backtracking of the best score can be found under `https://en.wikipedia.org/wiki/Smith-Waterman_algorithm#Example`.

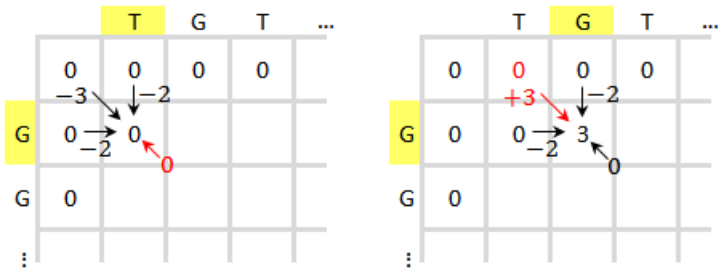

Figure 3.7.: Example for the computation of the score matrix (from [Wikb]).

3.11. Branch and Bound

Another popular technique for solving NP-hard optimization problems is Branch and Bound [LW66, Cla99].

In Branch and Bound algorithms the space of all solutions is viewed as a tree, where the leafs are the actual solutions and the inner nodes stand for partial solutions. In order to find the optimal solution one has to explore the tree of all solutions (branch). But because exploring all solutions will take too much effort, there is also a bound step. When visiting a particular node (partial solution) first the cost of all solutions in this branch of the tree is bounded (e.g. the minimal cost to build up a full solution from the partial one). The search does not continue in this branch of the search tree if the determined bound for a branch is worse[2] than the best solution found yet. This ensures that the best possible solution will be visited, while the number of visited solutions, that do not look promising, is limited.

3.11.1. Example: Traveling Salesman

As an example for this technique I will use the Traveling Salesman problem (TSP) [Coo15]. The Traveling Salesman problem is NP-Complete [Pap77].
An input for the Traveling Salesman problem is a weighted graph. One can think of a map with the distances between cities that a traveling salesman wants to visit. The question is, what is the best route so that the salesman can visit all cities. "Best" refers to the cost of the edges of the graph used - e.g. the time to travel. The salesman can start at any node and wants to visit all other nodes before he returns to his starting point[3].
Obtaining good solutions for this NP-hard problem is of great importance, especially in the logistics sector [Coo15].

One simple way to parallelize the process of finding a good solution using OpenMP is

[2]Here, I am thinking of a problem where one wants to minimize a cost function, but this technique can also be applied to problems where one want to maximize a goal function.
[3]Therefore the starting point of a roung trip does not matter.

to divide the search tree among the different threads. The score of the best solution discovered yet is shared among the different threads. The key is to guard the update of the best score (e.g. using the `omp critical` pragma), so that only one thread can update it at a time, to avoid race conditions (lost update).

A more detailed explanation of the algorithm and the usage of OpenMP within this case with the help of a listing can be found in Appendix A.6.

3.12. Construct Graphical Models

Graphical models such as hidden Markov models [Rab89], are based on graphical representations, e.g. a state transition graph.

A Markov model (or Markov chain) is a model of a stochastic process. The model is described by a set of states and their transition probabilities. Therefore the next state of the modeled process only depends on the current state (time independent Markov chain). In a hidden Markov model the state of the modeled process is hidden. Instead one can observe so called emissions. Each state has a probability distribution for these emissions. The goal is to find out the characteristics of the underlaying process when one can only observe the emissions. For example such models are used in speech recognition, where the states represent the words or syllables spoken, while the emmisions are the actual (possibly noisy) audio data [Rab89].

A hidden Markov model shares some similarities to other machine learning tasks and can often also be viewed as sort of a map-reduce task (Section 3.7).
For example the Baum-Welch-Algorithm to find the unknown parameters of a hidden Markov model has an Estimation-Step, which can be seen as the map part and a Maximization-Step, which can be seen as the reduce part [Wun01].

Therefore I will only take a closer look on the more general class of map-reduce tasks within this thesis.

3.13. Finite State Machine

Finite state machines are described as *embarrassingly sequential* in the Berkeley Report [ABC+06]. Although further research has revealed some ways of exploiting parallelism in finite state machines [LSEJ11, CAH+11, MMS14], the scalability of parallelism is often limited in this use cases.
For example [MMS14] showed that multi-core performance for Snort regular expressions or Huffman decoding gain no further speedup when utilizing more than eight threads. Others like [CAH+11] use SIMD (single-instruction multiple-data) capabilities of processors to achieve parallel processing.

Therefore, as this use case can not further benefit from the improved scalability of MPI yet, as it seems that the parallel capabilities of even a single modern processor cannot be fully exploited yet, this use case will not be discussed further in this thesis.

4. Different types of communication used

In this chapter I give a general overview over the communication patterns that will be used when the different use cases (Chapter 3) will be translated to MPI. Chapter 5 will then give a more in-depth explanation of the actual translation.

Each variable in an OpenMP program may have a different access pattern. In order to reflect this and allow for a proper translation of the different access patterns to MPI, I defined a new `#pragma omp2mpi comm [comm-mode]` directive. With this pragma the programmer may give hints to the translation process, which communication scheme should be used for a variable. The pragma has to be used directly before a declaration of one or more variables. It is then used to determine which communication pattern should be used. The new `omp2mpi comm [comm-mode]` pragma allows for the following communication modes to be used, which are explained in the following sections:

- `distributed`: This pattern distributes the elements of an array among the different MPI processes.
- `reading`: This pattern optimizes the variable for reading, which means that it is duplicated to all MPI processes.
- `master_based`: With this pattern the variable resides on the master process. All other processes need to use RMA-Operations to access it.
- `default`: This pattern is used when no `omp2mpi comm` is given. It is the same as `distributed` for arrays and `master_based` for a single value variable (like an integer).

As usual the pragma will just be ignored if it is not supported by the compiler. The implementation of the new pragma directive is explained more in detail in Section 5.2.

Variables are classified into array variables (pointers) of arbitrary dimension, single value variables (like a single integer or double variable) and structs. Struct variables are handled in a different way (described in Section 5.8). Therefore the usage of the `omp2mpi comm` pragma for a struct declaration will have no effect for the translation to MPI and will be ignored. The following sections describe the different communication patterns used for arrays and single value variables.

4.1. Distributed

The distributed communication pattern may only be used for arrays. As it makes no sense to "distribute" for example a single double variable between two ore more processes, the usage of `#pragma omp2mpi comm distributed` for a single value variable will be ignored and the default communication pattern (which is the master-based pattern described in Section 4.3) will be used instead.

For an N-dimensional array, elements that are of dimension $N-1$ will be distributed among the different MPI processes. Therefore each rank will be assigned to own a certain subset of $N-1$ dimensional array elements. This means for a 1-D array that each rank will own a subset of the array elements, for a 2-D array each rank will own a subset of the matrix lines, for a 3-D array each rank will own a subset of the cubes planes.
When one rank needs to access an element that is not owned by itself, it needs to use RMA routines to load/store this array element from/to the other owning rank. A detailed explanation of the implementation for this communication pattern is in Section 5.6.2.

This pattern reflects the fact that many applications distribute the data among the different threads so that each thread may work on a separate chunk of it. This pattern is used by default for array variables.

4.2. Reading

The reading pattern may be used for arrays as well as for single value variables.

For the reading pattern the data will be duplicated to every MPI process. Therefore any reading operations on the data can be done localy without the need of MPI communication routines. However, this means that an update (write operation) to the shared data has to be applied to all MPI processes (using MPI-RMA operations). Therefore, an update of the data is much more expensive than in the other communication patterns. The benefit is that reading the data is less expensive as no communication needs to take place. One can find a detailed explanation of the implementation for this communication pattern in Section 5.6.4.

This pattern reflects that some variables are mostly used reading while they are still shared among all threads (e.g. a variable that is checked after every iteration if the calculation should end). This pattern is not used by default but can be enabled by using `#pragma omp2mpi comm reading` before the declaration of the variables that should be shared this way.

4.3. Master based

The master based pattern may be used for arrays as well as for single value variables.

In this communication pattern only the master process stores the data. This means that all other processes will use MPI-RMA operations to read or write this data on the master process. A detailed explanation of the implementation for this communication pattern is in Section 5.6.3.

This pattern reflects that some variables are mostly used by the master process, for example a variable that is mostly used inside of OpenMP single regions. One can also think of such a variable as an output variable that is given to the master process. This pattern is used by default for single value variables and can be enabled by using `#pragma omp2mpi comm master_based` for arrays[1].

[1] Also `master` instead of `master_based` as the communication mode will be acknowledged.

5. Translation of OpenMP to MPI

In this chapter I will discuss the translation of OpenMP programs to MPI programs, that uses the communication types presented in Chapter 4.

Within this chapter the pass refers to the implemented LLVM pass that translates OpenMP to MPI, if no other LLVM pass is explicitly mentioned.

5.1. Overview of the translation process

The translation process based on [Ble17] is structured in six basic steps (also refer to Figure 2.2):

1. The Clang preprocessor plugin handles the `omp2mpi` pragma directives in the source code (see Section 5.2).
2. The source code is translated to LLVM IR by Clang.
3. Some basic LLVM passes such as `mem2reg` [mem18] are executed.
4. The implemented pass that translates OpenMP to MPI is executed.
   a) Analysis of the IR to find all parallel regions (refer to Section 5.3).
   b) Introduction of `MPI_Init` and `MPI_Finish` (refer to Section 5.3).
   c) Replacement of OpenMP pragma directives and library calls such as barrier or `get_num_threads()` (refer to Sections 5.4, 5.5 and 5.9).
   d) Insertion of communication for shared variables in each parallel region (refer to Sections 5.6 and 5.8).
   e) Verification of the resulting IR code.
5. Additional LLVM optimization passes are executed according to the specified optimization level. At least optimization level `O2` is required, so that all passes required for step 3 will be used.
6. The LLVM backend produces the actual machine code.

5.2. Pragma handling

In order to decide for the communication pattern to use for a specific variable, a new `#pragma omp2mpi comm [comm-mode]` is introduced via a Clang plugin. The idea behind this new pragma is explained in Chapter 4, while this section covers the implementation details.

When this pragma is encountered by the preprocessor, it annotates the next statement if a valid comm-mode was given. Clang allows to annotate variables using the compiler-builtin `__attribute__((annotate("example anotation")))`. Therefore the pragma handling plugin inserts this annotation, annotating a string literal corresponding to the communication mode that should be used. In the LLVM IR this annotation will show up with a call to the intrinsic `llvm.var.annotation` function, that takes four arguments [llva]:

1. The pointer to the annotated variable.
2. The annotated string literal.
3. The name of the source file where the annotation originates.
4. The line number of the annotation in the source file.

This intrinsic function has no other purpose and will be ignored by other optimizations as well as the generation of the resulting machine code [llva].
Therefore, the pass can find the annotated string literal from the call instruction to this intrinsic function, using it to decide which communication pattern should be used for a shared variable. The factory method, used to instantiate `SharedVariable` objects, will use the annotation in order to determine the subclass of the `SharedVariable` that will be constructed (refer Section 5.6.1).

The pass searches for this annotation to be defined in the function, where the `#pragma omp parallel` pragma is used. This is illustrated in Listing 5.1, where the annotation in the `main` function will be ignored (line 9), as this is no shared variable in the scope of `main`. Therefore, in order to recognize the annotation, it has to be done within the function where the parallel region is used (illustrated with the `tgt_func_with_annotation` in Listing 5.1). Note that in Listing 5.1 the function `tgt_func` is equivalent to `tgt_func_with_annotation`, as OpenMP basically will transform `tgt_func` into `tgt_func_with_annotation`, which means that the argument passed to the function is allocated at the stack again (see line 8 of Listing 5.1).

This must be done if an argument is passed by value. In order to share a variable, a pointer to the variable is needed, so that all threads can access this variable. This does account for arrays as well, although dynamically allocated arrays are pointers. If a pointer is declared shared, the value of the pointer is shared among all threads, which means that all threads should be able to change it (e.g. to increase the size of the array).

```
1  void tgt_func(int defined_in_main) {
2  #pragma omp parallel shared(defined_in_main)
3      { /* do something in parallel */ }
4  }
5
6  int main() {
7  #pragma omp2mpi comm reading
8      int defined_in_main = 42;
9
10     tgt_func(defined_in_main);
11 }
12
13 void tgt_func_with_annotation(int defined_in_main) {
14 #pragma omp2mpi comm reading
15     int defined_in_tgt = defined_in_main;
16
17 #pragma omp parallel shared(defined_in_tgt)
18     { /* do something in parallel */ }
19 }
```

Listing 5.1: Placement of the annotation of shared variables. The annotation in line 9 will be ignored.

In order to change a shared variable, its address is needed, so that the change is visible to all other threads in the shared memory.

The issue of the scope of the annotation is not important for global variables, as the scope of the annotation is global as well. For global variables the annotations are handled a bit differently in LLVM IR: Instead of a call to the intrinsic `llvm.var.annotation` function, a new global variable `llvm.global.annotations` is inserted, which is an array initialized with the annotations of each global variable in the same way as they would have been passed to the `llvm.var.annotation` function. Again this global annotations will be ignored by other optimizations as well as the generation of the resulting machine code.

5.3. Sequential regions

OpenMP is based on the Fork-Join concept (refer to Section 2.3), whereas in an MPI application all processes are initialized at the beginning and start calling the main function[1]. As the OpenMP user expects the application to have only one thread executing the sequential regions the translated programm will emulate this behaviour.

The pass will insert a conditional branch into the first block of the main function

[1]It is possible to spwan new processes in MPI (e.g. using the `MPI_Comm_spawn` function [mpi]), but this is not used in the current implementation of the translation.

(after the call to `MPI_Init`). So that only the master continues into the sequential part of the program. The worker processes will call a new function `worker_main`. In this function they will wait for a notification from the master (`MPI_Bcast`). When the master wants to enter a parallel region, he first broadcasts the ID of the region to enter, as the master may enter a specific parallel region based on the execution of the sequential part. Therefore, each parallel region will have a constant unique ID to identify it, which is assigned by the pass. Assigning a unique ID to each region is possible, as all parallel regions are known at compile time and branching into or out of a parallel region is forbidden by the OpenMP standard.

When the workers receive the broadcasted ID from the master, they will basically join the execution path of the master. First the communication information for all used variables needs to be initialized (see Section 5.6.2 arrays, Section 5.6.5 for single value variables or Section 5.8 for structs). Furthermore the values of the firstprivate variables are broadcasted by the master process. When all the setup is done, each process will call the `.ompoutlined.` function. This function is outlined by OpenMP in order to represent the parallel region [Ble17]. When a process returns from the parallel region it has to wait in at the barrier at first. This means that an explicit `MPI_Barrier` is inserted to represent the implicit OpenMP barrier at the join of the threads. After this join-barrier, the communication information may be freed (e.g. gathering a distributed array from all processes to the master (see Section 5.6.2)). The master continues executing the following sequential part, while the worker will wait for the next broadcast from the master.

As the exit of the program should also be performed by all processes it will have a unique ID too. At the exit the master will broadcast the exit-code of the program, so that each process will exit with the same code. Finally `MPI_Finish` is called, before all processes return from the main function.

5.4. OpenMP synchronization directives

5.4.1. OpenMP barrier

The usage of the OpenMP barrier pragma for synchronization of the threads is replaced with a call to `MPI_Barrier` by the pass. This also applies to the implicit OpenMP barriers after worksharing constructs.

5.4.2. OpenMP critical pragma

In order to properly translate the OpenMP critical pragma, the pass replaces it with the usage of an MPI mutex implemented by [Pre11].

The mutex is implemented with a list that is shared by RMA. In this list every process can indicate that it is waiting for the mutex (or currently holding it). When one process

wants to acquire the lock it first updates the waitlist, and then finds out whether other processes are also currently waiting. If others are waiting as well it will wait for a message that the mutex is now free to use for it. Otherwise it now owns the mutex.
If a process releases the mutex, it has to check whether other processes are waiting as well and then notify one of them. So the ownership over the mutex is transferred to the other process. For fairness, each process will start looking for other processes that may be waiting for the mutex starting at their own rank +1. This will result in the mutex passed around in a round robin fashion instead of favouring some processes like rank 0.

5.5. OpenMP worksharing directives

OpenMP offers several so-called worksharing directives, used to distribute the workload to the different threads. This section covers the translation of some of these worksharing directives to MPI.

5.5.1. OpenMP for loop

A commonly used feature of OpenMP is the `for` pragma directive, which allows to spread the work of a for loop among different threads, so that each thread will execute a certain subset of the loop iterations.
The LLVM OpenMP implementation will introduce an OpenMP for loop, that has a static schedule, with a call to `__kmpc_for_static_init_`. This function is responsible for calculating the iterations that each thread will execute. The pass will replace this OpenMP function with its own function. The translation of a non static schedule for the loop iterations is currently not supported. Therefore every MPI process can calculate the iterations it should execute on his own and no communication is needed. After the loop the process, that has executed the last loop iteration, will broadcast the value of all variables that have been declared `lastprivate` in the for pragma (using `MPI_Bcast`), in order to match the definition of the `lastprivate` clause.
The implementation of this feature is based on [Ble17].

5.5.2. OpenMP section pragma

For worksharing OpenMP also offers a section pragma, so that each thread will execute a different section of the code. Internally, the section pragma is represented as a for loop that iterates over the sections. Therefore, the translation of the for loop will also properly translate the sections pragma.

5.5.3. OpenMP single and master pragma

Single and master pragmas are used inside an OpenMP parallel region to indicate that the following block should only be executed by one thread of the team. Currently the pass treats the single pragma like the master pragma, as this eliminates the need for

communication and still conforms to the OpenMP standard. Therefore, only the master process will execute a single or master region.

5.5.4. OpenMP task pragma

When a new task is created with the `omp task` pragma, three steps are performed by OpenMP:

1. Allocation of a new task.
2. Storing the location of the shared and value of the private variables for the task.
3. Declare this new task using the `__kmpc_omp_task` function.

The pass will insert analogous operations in the MPI version, using a global dynamic window for the tasks.

1. Allocate a new task and attach it to the window.
2. Storing the location of the shared and value of the private variables for the task.
3. Add the task to a globally shared list of tasks.

The current implementation only allows variables to be shared with the new task only if they are defined as shared in the enclosing parallel region. Therefore each variable will get an unique ID. The task will read the ID instead of the pointer to the shared variable when entering the task. It will match the ID to the correct communication information, that can be handled like communication within every other parallel region (Section 5.6). A variable that is shared in the enclosing parallel region might be defined private for the task. As OpenMP will then create a distinct copy of the variable, there is no need for the pass to intervene in this behaviour.

The value of each firstprivate variable is copied to a new buffer, that is an element of the list of private variables. As a task may use an arbitrary number of variables of different types, the variables are managed as a linked list[2]. Each list member is attached to the global `MPI_win`. The pointer to the next list element is the same as for any other shared struct (Listing 5.9). When the task is called, first the list of firstprivate variables is traversed in order to load their values. The mapping of the variables, which variable of the task creator matches to which variable of the task, is done via the ordering of this variables in the list.

The list of available tasks is also shared in the same way. The pointer to the list head always resides at the same position on the master process so that it is accessible by RMA operations for all other processes. This means that at the initialization of the parallel region that has tasks, the pointer to the list head needs to be broadcasted once.

[2]So that each list member may have a different size to hold the data.

Each barrier and taskwait pragma in a parallel region that has tasks is replaced with a call to a new `mpi_sync_tasks` function[3]. For ease of implementation this function takes a flag as argument whether to perform a taskwait or a barrier functionality. The pass will insert a call with the correct value for the flag set.
Each process that enters the `mpi_sync_tasks` tries to take one task out of the list of available tasks and start working on it. If no further tasks are present and all processes have entered the `mpi_sync_tasks` function, all processes will synchronize using `MPI_Allreduce` to determine whether at least one process entered the `mpi_sync_tasks` as a taskwait. Then only the processes that entered a function as a taskwait will return and the other processes will see if there are new tasks available. If all processes entered the `mpi_sync_tasks` as a barrier, all processes will continue the execution of the program. As `MPI_Allreduce` implies a barrier, no explicit barrier is needed in this case.

5.6. Translation of different communication patterns

This section describes the translation for the different communication patterns introduced in Chapter 4 in detail. The first subsection explains the general design choices made, while the other subsections will each describe one particular communication pattern.

5.6.1. Design of the translation process for different communication patterns

In order to allow for an easy implementation of different communication patterns, each shared variable is represented by an instance of a subclass of the virtual `SharedVariable`. For each type of variable (arrays and single values) a virtual derived intermediate class exists. For each communication pattern there is a seperate class derived from the intermediate class. For example the `SharedArrayDistributed` class represents arrays that should be distributed among the processes (as described in Section 5.6.2). This hierarchy is shown in Figure 5.1. The `SharedUnhandeledType` class exists to model variable types that currently cannot be shared, such as stack allocated arrays or function pointer.
Each derived class has to implement the virtual functions that are part of the interface for the `SharedVariable` class. The most important part of this interface is shown in Listing 5.2.

First of all, there is a static factory method (line 4 of Listing 5.2), that will create new instances of the derived classes, depending on the type of the variable and the communication pattern that should be used. The process of deciding when to use which communication pattern was introduced in Chapter 4. The actual interpretation of the introduced `#pragma` directives is explained in Section 5.2 more in depth.

The `get_local_buffer_size` method (line 7 of Listing 5.2) returns the size of the local buffer needed to store the shared variable. For example this is equal to `sizeof(int)` for

[3]Also the implicit OpenMP barriers are replaced.

```
1  class SharedVariable {
2  public:
3      // factory that constructs a shared variable with their specific
       ↪ type
4      static SharedVariable *Create(llvm::Module &M, environment *e,
       ↪ llvm::Value *var);
5
6      // always used as the second arg for communicating functions:
7      llvm::Value *get_local_buffer_size(llvm::Module &M, environment
       ↪ *e);
8
9      virtual llvm::StructType *get_comm_info_type(llvm::Module &M,
       ↪ environment *e) = 0;
10
11     // which functions should this variable use for communication:
12     virtual llvm::Constant *get_comm_function_on_store(llvm::Module
       ↪ &M, environment *e) = 0;
13     virtual llvm::Constant *get_comm_function_on_load(llvm::Module
       ↪ &M, environment *e) = 0;
14     // init and finish a parallel region:
15     virtual llvm::Constant *get_comm_function_on_init(llvm::Module
       ↪ &M, environment *e) = 0;
16     virtual llvm::Constant *get_comm_function_on_finish(llvm::Module
       ↪ &M, environment *e) = 0;
17     // at synchronization point (e.g. barrier or taskwait):
18     virtual llvm::Constant *get_comm_function_on_sync(llvm::Module
       ↪ &M, environment *e) = 0;
19 };
```

Listing 5.2: Exerpt of the `SharedVariable` class declaration.

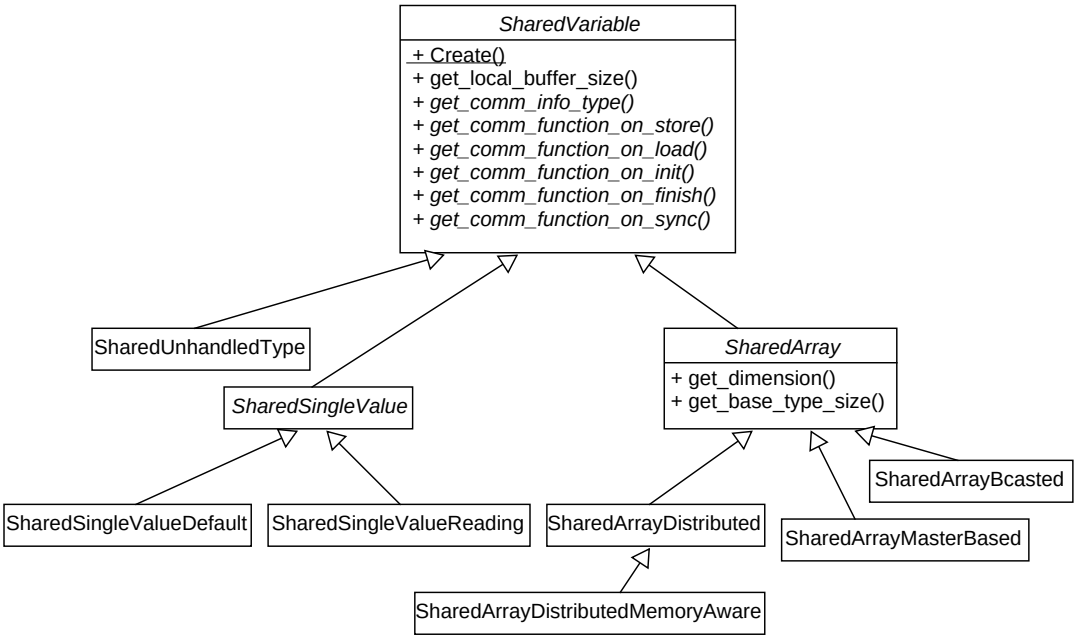

Figure 5.1.: Inheritance hierarchy

an integer variable or the size of a pointer for array variables. As the size of the variable does not depend on the communication pattern, this method is not virtual and implemented in the base class. As shown in Figure 5.1, the `SharedArray` class also offers the functions `get_dimension` to get the dimension of the array and `get_base_type_size`, which gives the size of one element of the array.
The size of the variable is also needed to build the communication info struct. The communication info struct is explained in the following subsection.

The virtual functions defined in lines 12 to 18 of Listing 5.2 are implemented by the specific subclasses to return a pointer to the specific function that is called in order to appropriately handle the load, store and synchronization of the variable as well as the initialization and finalization of a parallel region. If a nullpointer is returned, no function call will be inserted.

This allows for the code that does the actual insertion of the communication routines to be very generalized. It is summarized in Listing 5.3 for single value variables.

First the pointer to the variable will be casted to the C-type `void*`, which is `int8*` in LLVM IR (line 6 of Listing 5.3). The cast is done, so that the communication functions can take it as a `void*` rather then overloading the function for each type of variable. Therefore the communicating functions also need the size of the variable as the second argument (line 8 of Listing 5.3). For every instruction that uses the shared variable (line 10 of Listing 5.3) communication will be inserted if needed. For a store instruction

```
1  void insert_comm_single_value_var(Module &M, environment *e,
       ↪ ParallelFunction *microtask_obj,
2                                    SharedSingleValue *shared_var_obj)
3  {
4      IRBuilder<> builder(M.getContext());
5      builder.SetInsertPoint(get_init_block(M, e,
           ↪ microtask_obj->get_function())->getTerminator());
6      Value *comm_info_void =
           ↪ builder.CreateBitCast(shared_var_obj->value(),
           ↪ builder.getInt8PtrTy());
7      // the comm_info_struct is always passed as void* to the
           ↪ communication functions
8      Value *type_size = shared_var_obj->get_local_buffer_size(M, e);
9
10     for (auto *u : shared_var_obj->value()->users())
11     {
12         // check whether one should insert communication for this
               ↪ particular use of the shared variable
13         if (u != comm_info_void && is_comm_allowed(e, u) &&
               ↪ inst->getFunction() == microtask_obj->get_function())
14         {
15             if (auto *store = dyn_cast<StoreInst>(u)) // if it is a
                   ↪ store instruction:
16             {
17                 if (shared_var_obj->get_comm_function_on_store(M, e))
18                 {
19                     builder.SetInsertPoint(store->getNextNode()); //
                           ↪ insert after the store
20                     builder.CreateCall(shared_var_obj->
                           ↪ get_comm_function_on_store(M, e),
                           ↪ {comm_info_void, type_size});
21                 }
22             }
23             // else if it is a load instruction:
24             /* insert a call to the comm_function_on_load before the
                   ↪ load */
25         }
26     }
27     /* insert the call to the syncronization function on sync point */
28 }
```

Listing 5.3: Function that adds MPI communication to a shared variable of the single value type.

```
1 struct comm_info {
2     int local_value;
3     single_value_info comm_info_part;
4 };
```

Listing 5.4: Example of a communication information struct.

(line 15), after the store (line 19) a call to the correct cummunication function for this variable will be inserted (line 20), if a communication function was given (line 17). The load instruction will be handled analogously.
Other operations on a shared variable, such as an atomic increment, are currently not supported. Besides load and store instructions only call instructions that take the shared variable as an argument are supported as well[4], as long as the callee of the call instruction is also defined within the translated module (Refer to Page 52 for the case when the callee is not defined within the translated module).
Additionally, before any synchronization point - like a barrier - a call to the given synchronization function will be inserted.

This means that in order to implement a new MPI communication pattern one basically only needs to follow four simple steps:

1. Implement the new communication functions in the communication library (also refer to Page 51).

2. Introduce them to the pass so that it may insert calls to them. This is further explained in Appendix B.

3. Define and implement a new derived class from `SharedArray` or `SharedSingleValue`, depending on the type of variable. The technical details of defining this new class are outlined in Appendix B.

4. Adapt the factory method so that it will create instances of this new class (also refer to Section 5.2 and Appendix B).

Implementation of the communication functions

In order to allow an easy implementation of the communication functions, each shared variable is represented by a struct. The first member of the struct is always the local value of the variable. An example for an integer value is shown in Listing 5.4. The pass builds a struct type consisting of the local buffer and the communication information needed for the communication pattern of this variable. This is done by calling the `get_comm_info_type` function of the `SharedValue` class, which should return the correct communication information struct.

[4]Sharing a function pointer as a variable is not supported.

As shared values are always pointers in OpenMP, this allows to use the pointer to this struct just as a pointer to the variable, because loading from this (struct) pointer will load the first struct element, which is the value of the variable. This allows for the pointer to be passed to other functions that may not be defined in the translated module and therefore can not be translated to MPI. These functions can then work with the local copy of the variable. Also the code that needs to work with the variable does not have to be changed, except inserting new communication routines.

The communication functions can then access the other information stored in the struct in order to keep the local copy of the variable up to date. The functions that will use this additional information are implemented in a seperate module in C++. This allows for an easy implementation of new communication schemes, as there is no need to insert the calls to the MPI routines directly in the IR code. Instead a call to a library function is inserted, which uses the additional information held in the communication info struct to call the correct MPI routine. These communication functions are described in the following subsections.

The translated program however is not linked against this communication library. Instead the library is compiled to IR code, which is then inserted into the translated module by the pass. Functions that are not used are spared out in order to not increase the size of the translated module unnecessarily. This allows the LLVM optimization passes that were run after the translation to MPI to optimize the calls to the communication functions. Especially important is, that this allows to inline them, for example when they only consist of one MPI call.

This structure of the communication would even allow to change the communication pattern at runtime if this seems beneficial to performance. This can be achieved by also storing the communication pattern to be used in the communication information part. But as this will come at the overhead to decide in every communication call which type of communication should be used, this is currently not implemented. Besides the currently implemented new pragma directives, another idea to determine the communication type that should be used is that a profiler may determine the best communication pattern beforehand, so that it is known when finally translating the program to MPI.

Different communication patterns

The following subsections, describing the implemented communication patterns, are each structured in a similar way: At the beginning a general overview over the implemented behaviour is given. This is followed by a more detailed explanation of the implemented procedures for initialization and finalization of a parallel region as well as load, store and synchronization points like barriers.
Currently all communication patterns only support arrays that were continuously allocated on the heap.

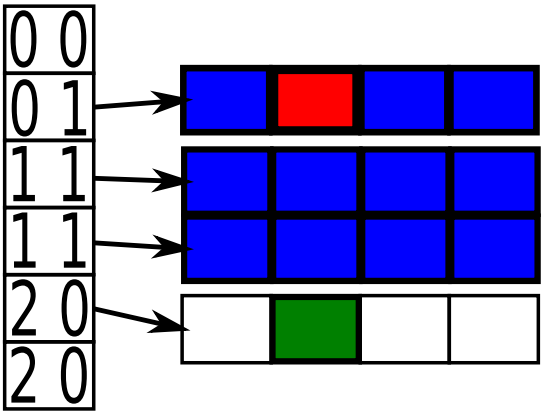

Figure 5.2.: Illustration of the 2D-array distribution for rank 1. On the left side the first array dimension is shown. The first number indicates the rank that owns this array line. The second number indicates if a valid local copy exists on the current rank. The valid data area is shown in blue, while red and green indicate example positions.

5.6.2. Distribution of arrays

Overview

As each process has its own memory space, large chunks of shared data such as arrays need to be distributed among the processes. Therefore each array will be distributed among all processes when entering a parallel region. This represents the distributed pattern introduced in Section 4.1.

The first dimension is distributed among the processes so that each process will get roughly the same number of elements - or lines, planes,... depending on the dimension of the array. The function for communication does not rely on a specific distribution of the array lines. Therefore, it will be fairly easy to implement other distributions where each process gets a different number of lines, as long as each process gets a continuous chunk of the data. The restriction that each process may only own a continuous chunk of the data is used when calculating the displacement to access an element.

Each process will hold the information if a local copy (cache) of an element currently exists within the own address space. In order to not change the code[5] to access an (multi-dimensional) array, each process allocates the full buffer for the first dimension. This is illustrated in Figure 5.2 for a 2-dimensional arrray.

The Figure 5.2 illustrates the view for rank 1. On the left side the first array dimension is illustrated. As for most multidimensional arrays in C/C++ the first dimension only stores the pointers to the deeper dimension. Figure 5.2 illustrates this through the arrows.

[5]Also refer to Section 5.7 for additional insight on why the code that accesses the array will be changed as little as possible.

When a line is currently not present on the current rank, the pointer is invalid. The numbers inside the first dimension are actually stored at a separate place, so that the original array structure is unchanged, especially for the owned chunk of the lines. The first number indicates which rank owns the associated line, while the second denotes if there is a valid copy of this line currently in the local memory. If the program accesses a line where no valid local copy is present, it may first need to allocate a new buffer to store that line and then load this line from the owning rank through RMA. This is illustrated for the red array position 1,1 in Figure 5.2. If the green element at position 4,1 should be accessed, the line has to be loaded from the owner (rank 2) again, as the local copy is currently not valid. Here no new buffer is allocated as the old one with the invalidated elements will be used again. If another blue element needs to be accessed, no further communication will take place, as there is a currently valid local copy, or it is an own element respectively (for lines 2 and 3).
This allows for all array indices to index the global array, so there is no need for an index transformation to the local array index.

As the programmer has to make sure no race condition occurs, the cached lines - that are owned by other ranks - will only be invalidated when a synchronization occurs. This minimizes the need for communication on data updates while still conforming to the OpenMP standard.

In general, as the code of accessing the array is not changed - only communication is inserted, most operations on the array elements should be safe. As the memory of the array is not continuous anymore, due to the distribution over multiple processes, iterating over the elements by just incrementing a pointer may be dangerous (also refer to Figure 5.2). Especially when using elements that are not owned by the current process. The translation assumes that an array is accessed the "standard" way, e.g. using `Matrix[i][j]=value` as the value for `i` is needed in order to check if the used line is owned by the current rank[6].

This distribution was tested with 1, 2 and 3-D arrays, but should also work on an arbitrary number of dimensions.

Communication info struct

Listing 5.5 shows the `array_distribution_info` struct. The information needed for distributed arrays are described as follows:

- The `MPI_Win` that will be used for RMA operations on the array (line 2).
- The upper and lower bound of the owned chunk of the array (lines 3 and 4).
- The dimension of the array (line 5).

[6] `(*Matrix)[j]` for `Matrix[0][j]` is also supported, as there might be no difference in LLVM IR for this statements.

```
1  typedef struct {
2      MPI_Win win;
3      int own_upper;
4      int own_lower;
5      int DIM;
6      int D1_count;
7      size_t D1_elem_size
8      int *NDIM;
9      int *location_info;
10     std::vector<long> *currently_cached;
11     long *upper_lines;
12     size_t elem_size;
13 #ifndef DESTROY_GLOBAL_ARRAY_IN_PARALLEL_REGION
14     void * orig_master_array_ptr;
15 #endif
16 #ifdef ARRAY_PROFILING
17     array_profiling_info *array_profiling_info;
18 #endif
19 } array_distribution_info;
```

Listing 5.5: Communication info struct for distributed arrays.

- The size of an element of the first dimension. This is used to get the communication size when communicating a whole `DIM`-1 dimensional object or to determine the displacement when accessing elements that are owned by another process (line 7).
- The size of each array dimension (unit is the number of elements). This is needed when allocating the local array buffers (line 8).
- The information which rank own which parts of the data (line 9) and whether a valid copy exists on the current rank.
- A list of indices that are currently cached from other ranks. This is used for optimization as this information may also be extracted from the `location_info` array (line 10).
- The first (global) index of the assigned chunk for each process, used for displacement calculation when accessing data from other ranks (line 11).
- The size of a single element, which is used for allocation and offset calculation (line 12).
- *optional:* The original array pointer on the master process if the original array should not be destroyed during the parallel region (line 14).
- *optional:* The pointer to the associated array profiling info, which can be used to allow for simple profiling of the array usage (line 17). This is described in Section 6.3.

Initialization procedure

The initialization procedure works as follows:

1. The master finds out the size of each dimension of the array and broadcasts this information to all worker processes[7].
2. Each process calculates which chunk of the data it will own.
3. Each process allocates a buffer for its assigned part of the array.
4. The data is distributed.

   For the distribution of large arrays the usage of `MPI_Scatterv` is not sufficient, as the displacement vector consist of integer, so displacements larger than 2GiB are not representable. Therefore the distribution is done by each process calling `MPI_Get` for each assigned line to get the data from the master process. This means that in the current implementation each N-1 dimensional object (line) may not be larger than 2GiB, as the sendcount in `MPI_Get` is an integer variable.
5. *optional:* The original array will be freed on the master process.
6. A new MPI window is created, where each process exposes their chunk of the data for RMA operations.
7. The information which line belongs to which process and whether the current process has a valid copy is initialized.
8. *optional:* The array profiling information is initialized (see Section 6.3).
9. An exposure epoch on the MPI window is started for all processes.

The last step of starting an exposure epoch right away is done, so that the processes are constantly exposed to writing operations, which means that there is no need to call `MPI_Win_lock` (or unlock) for a writing operation. This is beneficial to the performance of writing operations, as there is no need to wait for the RMA operation to complete on the target rank.

Procedure on load

1. Check if a currently valid copy of the accessed element already exists.

   If a valid local copy exists - e.g. because the accessed element is part of the owned chunk - there is nothing more to do and the routine is finished.
2. *Only if it does not exist yet:* Allocate a new buffer to hold the accessed line.

[7] The process of finding the array size from memory allocation size was based on [Ble17] .

3. Load the accessed line from the other rank using `MPI_Get`.

4. Ending the RMA exposure epoch in order to wait for the loading to complete.

5. Re-open the exposure epoch again for future writing operations.

In order to minimize the need for communication a whole line[8] is loaded at once (step three of this procedure). Therefore subsequent loads from the same line will only result in one MPI RMA operation.

Procedure on store

Writing onto lines owned by another rank is performed on a per element basis. This means that if the accessed element is not owned by the current rank, the procedure will initiate a call to `MPI_Put` to apply the writing operation to the original data.

Procedure on synchronization point

1. End the exposure epoch, so that all RMA operations finish. This will enforce a consistent memory view for future RMA operations.

2. Start the next exposure epoch.

3. Invalidate all copies of data from another process.

In order to minimize the need for communication, there is no checking, whether the original data on another rank has changed. The local copy will be invalidated at every synchronization point. The reasoning behind this is, that the synchronization (for example using a barrier) is done in order to complete changes to the data. Therefore this communication pattern just assumes that simply loading the data again from the remote rank does require a comparatively smaller overhead than determining whether the remote original data really has changed. The next loading procedure need to load the data again from the remote rank, even if it might not have been changed, but as the synchronization itself is seen as the indication that the data has changed, this unnecessary re-loading should not occur that often.

Finalization procedure

1. The RMA exposure epoch is closed for all processes, causing all RMA operations to finish.

2. *optional:* The array profiling information is gathered (see Section 6.3).

3. *optional:* The master process allocates the full array again.

4. The data is gathered to the original master array.

[8] "line" here is referring to a full $N-1$ dimensional object.

```
1 typedef struct {
2     MPI_Win win;
3     int DIM;
4     int D1_count;
5     size_t D1_elem_size;
6     int *NDIM;
7     size_t elem_size;
8 } master_based_array_info;
```

Listing 5.6: Communication info struct for master based arrays.

5. The `MPI_Win` is freed.
6. All memory needed for the distribution (including the buffer for the assigned chunk of the array) is released.

Variation

A variation of distributed arrays may also be used (`distributed_memory_aware`). The variation differs in the invalidation of cached array lines. The memory aware variant will free the local buffers every time when they are invalidated. The advantage is that less memory will be used, which is especially important when irregular communication patterns will be used, where one line may not be used again. But this comes at the cost of reallocating the local buffer again when a regular communication pattern will be used where the same line is accessed very often. Additionally the reallocation of the local buffers might lead to heap fragmentation.

5.6.3. Master based arrays

Overview

For master based arrays, only the master process holds the real data of the array. They are supposed to be used when the array is mostly read by the master process. This type of arrays can be viewed as an output array where all other processes write directly to the master process. One can also see this as a special case of a distributed array, where the master process owns all data, while the other processes have no chunk assigned.

Communication info struct

Listing 5.6 shows the communication information struct, that holds all information needed for master based arrays:

- The `MPI_Win` used for RMA operations (line 2).
- The dimension of the array (line 3).
- The number of elements in the first array dimension (line 4).

- The size of an ($N-1$ dimensional) element in the first array dimension, used for offset calculation when writing to the master (line 5).
- A vector containing the size of each array dimension (line 6).
- The size of each of the arrays elements (line 7).

This is a subset of the distributed array information. As the next subsections show, this also applies to the communication procedures, as this pattern may be viewed as a special case of distributed arrays, where the master process does own all the data. This will allow for an easy adaptation of this communication pattern in the future, as the code is not that specialized to only support arrays that are based on the master process.

Another important note is, that the local array buffer for the worker processes will be allocated in the same way as for distributed arrays. This means that every $N-1$ dimensional element for the first dimension will be allocated separately on the worker processes if at least one element of this data should be accessed. Although for this pattern it would be sufficient to just hold a buffer of one element at the worker processes, this will allow for more concurrent RMA operations. If there would be just a buffer with the size of one element, before every further write to this buffer, it has to be assured, that the RMA operation corresponding to the previous write to the array has been finished, so that the buffer may be reused. If there is more buffer space, there is no need to wait for previous RMA operations to complete, as it is the programmers responsibility to avoid race conditions.

Initialization procedure

The initialization procedure is basically a subset of the procedure outlined above for distributed arrays:

1. The master finds out the size of each dimension of the array and broadcasts this information to all worker processes[7].
2. A new MPI window is created, in which the master exposes the array for RMA operations by other processes. The worker processes do not expose any memory.
3. An exposure epoch on the MPI window is started.

Procedure on load

The procedure on load is quite similar to the procedure for distributed arrays. The only difference is that there is no check whether a valid copy of the accessed data already exists. This is spared as the communication pattern is intentionally not optimized for the worker processes to read the data.
If the master process access the array, no special procedure will take place as it can access its local original data.

```
1 typedef struct {
2     MPI_Win win;
3     int DIM;
4     size_t elem_size;
5 } bcasted_array_info;
```

Listing 5.7: Communication info struct for duplicated arrays.

Procedure on store

Also the procedure on store is quite similar to the procedure for distributed arrays, except that all array lines are owned by the master.

Procedure on synchronization point

Finally on a synchronization point the exposure epoch is ended and re-opened again, in order to wait for all outstanding RMA operations to complete.

Finalization procedure

At the finalization, the exposure epoch will be ended in order to complete all ongoing RMA operations. Then the `MPI_Win` will be freed, as well as any local buffer on the worker processes.

5.6.4. Duplicated arrays

Overview

Duplicated arrays will be copied to all processes. This allows fast reading from the arrays. But since an update to the array has to be applied to every rank, this comes at the tradeoff, that writing to a duplicated array is more expensive. Additionally, as each process will store the complete array, also more memory is needed compared to distributed or master based arrays.

Communication info struct

The communication info for duplicated arrays, shown in Listing 5.7, consists of the following information:

- The `MPI_Win` used for RMA Operations (line 2).
- The dimension of the array (line 3).
- The size of an element of the array (line 4).

In contrast to distributed and master based arrays, there is no need to keep the information about the sizes of the different array dimensions past the initialization procedure. The displacement of an array element will be the same on all ranks, as all processes have the same memory layout.

Initialization procedure

1. The master finds out the size of each dimension of the array and broadcasts this information to all worker processes[7].
2. The worker processes also allocate space for the array.
3. The array is duplicated to every process using `MPI_Bcast`.[9]
4. The `MPI_Win` is created in which every process exposes the array to RMA operations from other processes.

One difference to the other communication patterns is that no exposure epoch is started right away. This is done, because this pattern is optimized for reading and there is no need for RMA operations on reading.

Procedure on load

When data from a duplicated array is loaded, no further operations need to be made, as each process will just load the data from its local duplicate.

Procedure on store

When some process needs to write data to a duplicated array, the write operation will be applied to all other processes as well. Therefore the writing process will start an exposure epoch on the MPI window (using `MPI_Win_lock_all`). Then the process initializes the RMA operations to write the data to all other duplicates (`MPI_Put`), before closing the exposure epoch again (`MPI_Win_unlock_all`).

The ending of the exposure epoch will wait for all initialized RMA operations to finish, so that all duplicates will be updated before proceeding with the further execution of the program. As the assumption for this communication pattern is, that writing to the array is not an operation encountered regularly, this implementation will prefer to have the additional overhead when writing, rather than an additional overhead when synchronizing (see below).
Additionally this may increase the consistency of a writing operation as ending the exposure epoch will wait for the operation on all ranks to complete before proceeding. One may encounter inconsistent duplicates when a lost-update race condition occurs. In the case when different ranks loose different versions of the update, this may lead to different "duplicated" data. Nevertheless it is still the responsibility of the programmer to avoid race conditions such as lost update.

[9] `MPI_Bcast` is called several times, if the array is larger than 2 GiB.

```
1 typedef struct {
2     MPI_Win win; // Win used for RMA operations
3 } single_value_info;
```

Listing 5.8: Communication info struct for single value variables.

Procedure on synchronization point

At a synchronization no special procedure is done, as the implementation of the store already waits for completion of the RMA operations. Therefore, there is no additional overhead when synchronizing the processes, especially when no writing operations take place.

Finalization procedure

On finalization, after the `MPI_Win` has been freed, the worker processes will release the memory occupied by the duplicated array. There is no need to gather the data to the master process as it still has its (original) duplicate.

5.6.5. Single value variables

Two communication patterns are also supported to use for single value variables: master based (Section 5.6.3) and duplicated (Section 5.6.4). They are basically the same as for the array variables, but much simpler in implementation, as only one element is involved.

The communication information struct for both patterns is shown in Listing 5.8. One only needs the information about the `MPI_Win` that is used for this variable to expose it for RMA operations by other processes. The size of the variable is not needed, as every communication function will take this information as the second argument.

5.7. Optimizing access to shared arrays in loops

Arrays are often accessed within for loops. Therefore I will place a special focus on discussing optimization of for loops that access arrays which are distributed in the way Section 5.6.2 describes[10].

The restrictions on the loop that may be optimized in the way described in this section essentially reflect those, which the OpenMP standard defines for the usage of the for pragma. Additionally the accessed array line may only be dependent from the loop index by a constant offset, or not dependent on the loop index at all.

A major disadvantage of the distribution of shared arrays is that before every access the

[10] As in Section 5.6.2, this section discusses arrays of arbitrary dimension, while I will use the term "line" to refer to a $N-1$ dimensional element of the array.

implementation has to check whether the accessed line is valid on the current process and load the line from another rank if necessary. This will impose a conditional branch in each loop iteration. Therefore I eliminated as much of these checks as possible without altering the semantics of the program. There are two main cases:

If the index of the accessed line is independent from the loop index[11], the whole array line will be loaded before the loop (also see Section 5.6.2). So it is checked just once if the used line is valid on the current process. Instead of a traditional optimization of conditions in loops, where the condition is checked before the loop and the program will branch into a loop where the condition is set true or false respectively, there is no need to duplicate the loop code in my optimization. The load of one array line is just moved out of the loop. As the store is currently done on a per-element basis, this optimization is not possible for stores to the array. The pass offers the functionality to group the conditions for all stores together so that the loop code may only be doubled instead of growing exponentially for each store when using the traditional optimization to extract conditions from loops.

If the index of the accessed line only depends on the loop index by a fixed offset, the loop is split in three parts:

1. Iterations that access lines before the lines owned by the current rank.
2. Iterations that only access lines that are owned by the current rank.
3. Iterations that access lines after the lines owned by the current rank.

This makes sure that there is no need for any MPI communication routines in the second part. Therefore all calls regarding communication for the used arrays are eliminated in this part. If multiple arrays are used the second part is the subset of loop iterations where the accessed lines for all used arrays are owned by the current process. This optimization requires that each process owns a continuous chunk of the data, so that the boundaries of the three parts may be calculated. The optimization will double the loop code, as one version - with communication - is required for part one and three and another version - without communication - is needed for part two.
The removal of MPI communication routines for the second part allows the compiler to apply more advanced optimizations to the loop (e.g. to vectorize it), as there is no more difference to a loop only on local data anymore.

See Section 6.3 for a discussion of the impact on the applications performance the discussed optimization has.

[11] E.g. because the loops iterates over a inner dimension of the array.

```
1 struct mpi_struct_ptr
2 {
3     int rank;
4     MPI_Aint displacement;
5     size_t size; // size of target struct
6     // MPI_Win win;
7 };
```

Listing 5.9: Struct to share the pointer to a struct.

5.8. Handling of struct pointers

In OpenMP all memory is shared by default. Therefore destinations of pointers are always shared if two threads have the same pointer value even if the pointer itself is declared as a private variable in an OpenMP parallel region. In C/C++ one can build many linked datastructures like a search tree with structs that contain pointers. This section describes how the pass deals with this sort of shared datastructures.

Structs are handled analogous to the general design choices explained in Section 5.6.1. This means that for each struct a new `comm_info` struct will be used, which contains the original struct as the first element, followed by additional communication information (refer to Section 5.6.1 and Listing 5.4). The communication information part is shown in Listing 5.9. It contains all information needed to share a struct: namely the rank that holds the data, the size of the remote struct as well as the displacement to access the struct within the given `MPI_Win`. The `MPI_Win` itself is not part of the struct. It is instead a global variable, as the pass currently uses the same MPI window for all structs. This global window is a dynamic window and new memory will be attached if a process allocates a new struct. If a struct contains a pointer to another struct (like in a tree structure for the child elements), the communication information part will also have an additional `mpi_struct_ptr` member that holds the corressponding information for the linked struct.

Analogous to Section 5.6.1 instead of a pointer to the shared struct, a pointer to the `comm_info` struct will be used. This changes the allocation size of the original struct, as the `comm_info` struct is larger than a pointer. Therefore all allocations of a struct have to be changed. Also struct allocations within the serial region of the code are affected, as these allocated structs might be used shared later. Starting at the allocation of a new struct, all uses of the resulting struct pointer are changed, but also other originations of a struct pointer are handled:

```
1 builder.SetInsertPoint(inst);
2 std::vector<Value *> args = {void_local_buffer,
     ↪ ptr_to_mpi_ptr_struct, e->global->struct_ptr_win};
3 builder.CreateCall(e->functions->mpi_load_shared_struct, args);
4 builder.SetInsertPoint(inst->getNextNode());
5 builder.CreateCall(e->functions->mpi_store_to_shared_struct, args);
```

Listing 5.10: Handling the usage of a shared struct.

- Allocation of a new struct.
- Passing a struct pointer as a function argument.
- A function returning a struct pointer.
- Reading a struct pointer from another struct[12].

For all this origination points the usage of the shared struct will be replaced in a similar fashion showed in Listing 5.10. The instruction (called `inst` in Listing 5.10) that uses the shared struct contents is performed on the local struct buffer. A new local buffer is allocated on the stack for each origination of a new struct pointer. At first, the remote struct content needs to be fetched (line 3 of Listing 5.10). Then the operation is performed on the local struct buffer. After the instruction (line 4) the remote struct content needs to be updated (line 5 of Listing 5.10). The `mpi_load_shared_struct` function will be called with the pointer to the local struct content buffer (first part of the `comm_info`), the pointer to the `mpi_struct_ptr` struct (second part of the `comm_info`) as well as the global `MPI_Win` used for all structs. The function then uses the values in the `mpi_struct_ptr` struct to call `MPI_Get` (see Listing C.1) to fetch the most recent remote struct content. `mpi_store_to_shared_struct` will then call `MPI_Put` in order to update the remote struct content.

Arithmetic operations on a pointer to a struct are not allowed. As each process has its own address space, the resulting address may not be valid on all ranks[13]. Therefore only equal (`==`) or not equal (`!=`) comparisons of pointers are allowed, as other comparisons may yield different results on different ranks. This comparisons for (in-)equality will be changed by the pass, so that pointers are equivalent if they point to the same struct, which implies that the pointer destinations reside on the same target rank.

[12] E.g. load one of the children in a tree datastructure.

[13] This does not include operations that are needed in order to access the struct members. Such operations are not inserted as arithmetic operations by the programmer and therefore not considered "arithmetic" in this context.

5.9. OpenMP reduction clause

As shown in Listing 5.11 the reduction clause is handled by the LLVM OpenMP implementation as follows:

1. Each thread allocates an array in which the pointers to the local variables that should be reduced are stored (line 4 in Listing 5.11).

2. Each thread passes this array, in which the pointers to the local variables are stored, to the `__kmpc_reduce_nowait` or`__kmpc_reduce` function (lines 7 to 10 in Listing 5.11).

3. Additionally the `__kmpc_reduce_nowait` or `__kmpc_reduce` function is given a function pointer to a function inserted by OpenMP (`.omp.reduction.reduction_func`), which performs the reduction on the variables.

4. This `.omp.reduction.reduction_func` function takes two pointers to the array of pointers to the local values and then performs the reduction for all these values. The result of the reduction is always stored to the local variables of the first thread (lines 29-40 in Listing 5.11. The summation operation performed is in line 38).

5. The `__kmpc_reduce_nowait` or `__kmpc_reduce` function then calls this `.omp.reduction.reduction_func` function with the correct parameters so that it will be called on each thread's data.

6. As the last step the master thread will combine the reduction result (which is now stored in its local variables) with the initial value stored in the global[14] variable, where the program now finds the reduction result (lines 11 to 21 in Listing 5.11).

The Listing 5.11 does not show the `%.omp.reduction.case2` block. This block is executed, if OpenMP should perform an atomic reduction operation. Therefore it is basically the same as the `%.omp.reduction.case1` block except that an atomic addition (`atomicrmw add`) is used.

In order to translate this to MPI, the pass looks for a call to the `__kmpc_reduce_nowait` or `__kmpc_reduce` functions. If such a call is found, for each variable that is part of this reduction two pieces of information must be collected:

- The location of each of the variable has to be found. Therefore the pass will know the datatype of the variable (line 3 in the example of Listing 5.11).

- The operation performed on this variable has to be extracted from the `.omp.reduction.reduction_func` function. The operations performed on distinct variables may be different, if the user specifies multiple reduction clauses (line 38 in the example of Listing 5.11).

[14]Here global refers to the visibility to all threads not to the actual scope of the variable.

```
define internal void @.omp_outlined.([...] , i32* %reduce) {
entry:
  %reduce1 = alloca i32, align 4
  %.omp.reduction.red_list = alloca [1 x i8*], align 8
  %1 = bitcast i32* %reduce1 to i8*
; Do the actual calculation, where the partial sum %reduce1 is
   ↪ computed
  %2 = bitcast [1 x i8*]* %.omp.reduction.red_list to i32**
  store i32* %reduce1, i32** %2, align 8
  %3 = bitcast [1 x i8*]* %.omp.reduction.red_list to i8*
  %4 = call i32 @__kmpc_reduce_nowait([...], i8* %3, void (i8*, i8*)*
     ↪ @.omp.reduction.reduction_func, [...])
  switch i32 %4, label %.omp.reduction.default [
    i32 1, label %.omp.reduction.case1
    i32 2, label %.omp.reduction.case2
  ]

.omp.reduction.case1:
  %5 = load i32, i32* %reduce, align 4, !tbaa !2
  %6 = load i32, i32* %reduce1, align 4, !tbaa !2
  %add = add nsw i32 %6, %5
  store i32 %add, i32* %reduce, align 4, !tbaa !2
  call void @__kmpc_end_reduce_nowait([...])
  br label %.omp.reduction.default

.omp.reduction.default:
; end of parallel section
  ret void
}

define internal void @.omp.reduction.reduction_func(i8* %0, i8* %1) {
entry:
  %2 = bitcast i8* %1 to i32**
  %3 = load i32*, i32** %2, align 8
  %4 = bitcast i8* %0 to i32**
  %5 = load i32*, i32** %4, align 8
  %6 = load i32, i32* %5, align 4, !tbaa !2
  %7 = load i32, i32* %3, align 4, !tbaa !2
  %add = add nsw i32 %7, %6
  store i32 %add, i32* %5, align 4, !tbaa !2
  ret void
}
```

Listing 5.11: Example of OpenMP reduction clause in LLVM IR.

With this information the pass can insert a call to `MPI_Allreduce` with the correct `MPI_Datatype` and `MPI_Op` if this operation is predefined by MPI (e.g. `MPI_SUM` if the user specifies to use the operator + for reduction). There are two operations left that are defined in the OpenMP standard but not in the MPI one:

- `^` (bitwise xor): The pass will insert a new function that will perform the bitwise xor and treat this function as a user defined reduction function (see below).
- `-` (subtraction): Subtraction reduction is intended to be used to determine the sum of negative values. The OpenMP standard states that the subtraction reduction is equivalent to the summation reduction [Ope13]. Therefore the pass will use the `MPI_SUM` operation in this case.

Newer versions of OpenMP (version 4.0) also allow user defined reductions [Ope13]. User defined reductions are declared using the `omp declare reduction` pragma. Then the LLVM OpenMP implementation will insert an `.omp_combiner.` function which will perform the reduction according to the user declaration. The `.omp.reduction.reduction_func` will therefore call this `.omp_combiner.` in order to perform the user defined reduction.

The pass will add a wrapper function for the call to `.omp_combiner.`, so that the `.omp_combiner.` can take the arguments with their specific type while the type of communication buffers in MPI is always `void*`. The wrapper function itself has the signature of a `MPI_User_function`. The wrapper function ensures that the `.omp_combiner.` will be called in the correct way, as the semantic of an `MPI_User_function` is that the operations result should be stored in the second argument while the `.omp_combiner.` will store it in the first argument. Therefore the reduction operation must be commutative as no correct translation to MPI is done otherwise. The additional arguments (`int * len, MPI_Datatype *`) of the `MPI_User_function` will be ignored as the `.omp_combiner.` does not use them. Later optimization passes will then inline the call of the `.omp_combiner.` into the new wrapper function[15].

The pass will then insert a new global variable to store a `MPI_Op` handle for this user defined reduction. After the initialization of MPI the pass will insert a call to `MPI_Op_create` using the wrapper function as the `MPI_User_function` parameter, in order to define the user defined reduction operation - analogous to its declaration with the `omp declare reduction` pragma. Also a call to `MPI_Op_free` is inserted before the call to `MPI_Finalize`. The pass will use this newly defined `MPI_Op` in order to insert a correct call to `MPI_Allreduce` where the reduction should be performed.

This translation of user defined reduction operations is not allowed for the reduction of struct types, as it is not allowed in MPI that an `MPI_User_function` uses MPI communication routines. But there might be the need for MPI communication routines when

[15]The `.omp_combiner.` has the `alwaysinline` flag set from the OpenMP implementation. So there is no reason to change this in the OpenMP to MPI translation pass.

reducing structs that have pointers within them. Therefore instead of inserting a call to `MPI_Allreduce`, the pass will insert a call to a newly implemented `mpi_manual_reduce`, that does not forbid the usage of MPI communication routines within the reduce operation. Instead of `MPI_Op` the new function therefore takes a poiner to the wrapper function (reduce operation) as an argument to perform the reduction similar to `MPI_Allreduce`[16]. The implemented `mpi_manual_reduce` function uses `MPI_Send` and `MPI_Recv` to reduce the values with a tree communication structure. As the reduction is performed as a tree, the order in which the reduction operations are performed is not deterministic in that sense, that it depends on the number of processes.

The reduction of pointer variables is not supported. As every process has its own adress space, it does not seem reasonable to me to combine pointers in a distributed memory system[17]. Therefore the programmer should pass the values that should be reduced to a user defined reduction function and not the pointer to this values.

[16]Therefore also the declaration of this reduction operation with `MPI_Op_create` will be skipped when reducing struct types.

[17]The reduction of structs that contain pointer is allowed, see Section 5.8, as long as the pointer is not used in arithmetic operations.

6. Evaluation

In this chapter the performance of the developed translation to MPI (Chapter 5) is evaluated regarding the different use cases presented in Chapter 3.

6.1. Test system

In order to evaluate the presented translation of OpenMP to MPI (see Chapter 6), the cluster from the research group of Scientific Computing (WR cluster) has been used.
The WR cluster is a system used for research and teaching. There are different kinds of nodes. I used the nodes which are based on 2 Intel(R) Xeon(R) CPU X5650 @ 2.80GHz with 6 cores each and 2 threads per core. They are equipped with 12 GiB RAM and operated by Ubuntu 16.04.5 with Linux Kernel version 4.4. The nodes are connected with Gigabit-Ethernet (1GbE) and the cluster uses the distributed Lustre filesystem and SLURM for batch queueing. They use the MVAPICH2-2.2 implementation.

6.2. Methodology

This section describes the measurement methods used to evaluate the performance of the translated MPI programs for the different cases.

The runtime of a test case was measured with help of the `gettimeofday` method [get]. Measured were the actual execution of the algorithms (as shown in Appendix A), but not the initialization of the input data that has been done serially. These measurements include the cost of the initialization of a parallel region. This implies that the actual initialization of the MPI processes (namely the call to `MPI_Init`) was not included in the measurements. Each measurement starts after the master process initialized the data, but before it is distributed to the different processes, so that the cost of the data movement for the initial distribution is included. This will model the fact that such algorithms are often used as part of a more complex application.
For the measurement of the runtime the programs were run at least three times for each configuration or more often if there was a significant variance. The figures in this chapter show all measured datapoints, instead of giving mean and standard deviation for each configuration.

The memory consumption was measured with the Valgrind tool massif [Val17]. The memory usage is summed over all MPI processes. As one is curious about the fact how much memory a program needs in total, I only considered the peak memory usage, as this represent the amount of memory a system needs to have available for a program to work properly.

The measured memory usage includes the memory used for heap as well as the memory used for the stacks, although the amount of memory needed for the stacks is not significant in most cases. Therefore most figures showing the memory usage do not distinguish between heap and stack memory. Instead they will denote which part of the memory is needed for the main data structure the algorithm needs (foremost the input data).
If applicable, the MPI version will also show the memory overhead that occurs during the initial distribution of the data, when the local buffers on the worker processes are allocated but the original memory on the master was is freed yet (refer to Section 5.6.2). Therefore, during the initial distribution (and final recombination) the amount of memory needed for the data is twice as high for a short time. See Section 8.2 for a suggestion how this double memory consumption may be avoided in the future. The remaining memory usage is then classified as overhead.
For the measurement of the memory consumption, the program's memory usage was profiled at least three times for each configuration. For further analysis the maximum memory usage that was recorded was used.

Memory consumption is also one limiting factor for the evaluation. As the data is initialized within the sequential region of the program (typical for an OpenMP application in my experience), it has to fit into the main memory of one node. This limits the evaluation of the weak scaling behaviour, as problem sizes that do not fit into the main memory of one node cannot be used in the current version of the translation. Therefore I propose to also distribute the data within the sequential part of the program (as explained in Section 8.2) in order to fully exploit the fact that more nodes have more main memory available.

Note that as one node of the test system has 12 CPU cores with two threads per core (refer Section 6.1), the OpenMP versions were executed using up to 24 threads, while the MPI versions were tested with up to 48 processes, that were spread among two nodes. Measurements with not more then 24 MPI processes were executed with all processes residing on the same node.
As one goal of this thesis is to determine if a translation to MPI is a good solution when dealing with larger problem sizes, the weak scaling behavior of the cases was measured with up to 240 MPI processes (spread among up to 10 nodes). For the strong scaling behavior it is not useful to utilize that many processes, as the benefit of using more parallelization decreases with every new CPU core utilized (compare Amdahl's law [Lud17]).

One additional remark is, that the translated MPI programs use MPI one sided communication. Therefore, the performance of the translated programs may be better on a

system that uses a low latency interconnect hardware, such as InfiniBand, instead of the Gigabit-Ethernet (1GbE) connection which test system uses (see Section 6.1).

6.3. Optimizing access to shared arrays in loops

The impact of the optimization discussed in Section 5.7 is shown in Figures 6.1 and 6.2. One can see that the optimization has a very positive influence on the translated MPI version of the program.

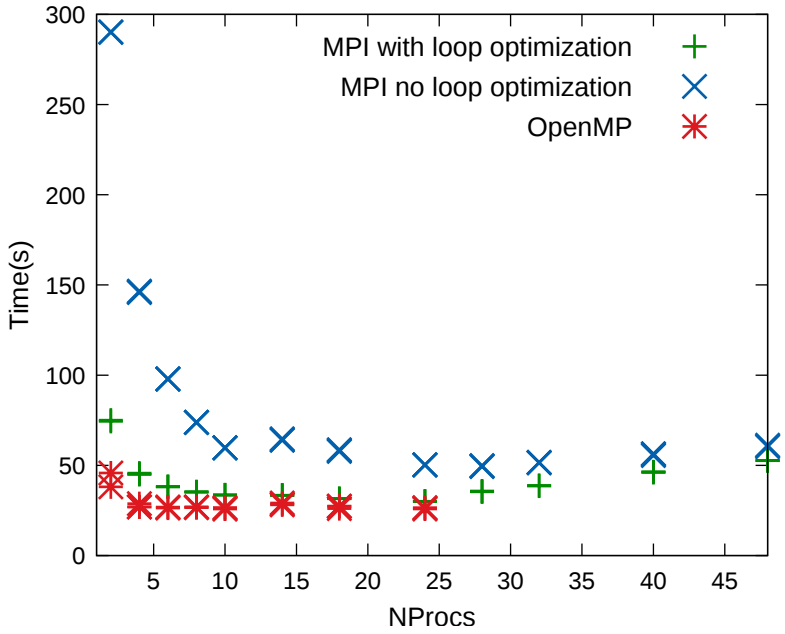

Figure 6.1.: Performance impact of the optimization discussed in Section 5.7 when solving a linear equation system (refer Section 3.1).

The exact influence depends on the considered use case though. For the case of a structured grid (Section 3.5) the performance gain shown in Figure 6.2 is not as significant as for linear algebra (Section 3.1) as shown in Figure 6.1. This is the case, because in the linear algebra use case the communication can be avoided nearly entirely as each process only operates on its own rows (refer Appendix A.1). Whereas communication of halo lines is needed in the case of a structured grid. As illustrated in Figure 3.5 communication has to take place when one process accesses the lines of another process at the boundary region between two processes. Therefore the performance gain from this optimization is strongly influenced by the amount of communication that can be eliminated.

In order to assess how much communication might be eliminated, the pass also supports to profile the usage of a shared array, collecting basic information such as how often a remote array line is accessed compared to an owned one. However the discussed optimization distort the collected data as the accesses where communication has been

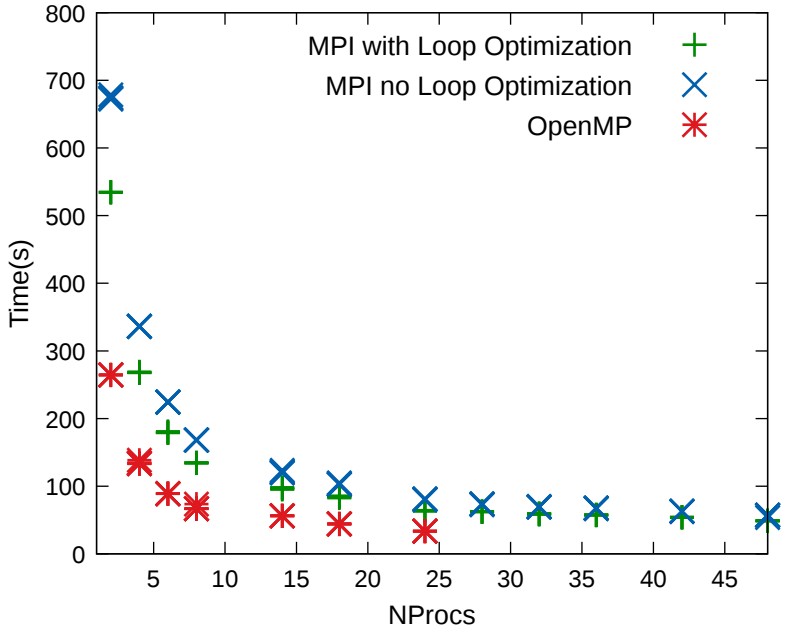

Figure 6.2.: Performance impact of the optimization discussed in Section 5.7 when solving a PDE (refer Section 3.5).

securely eliminated are not counted anymore.

6.4. Linear equation

Figure 6.3 shows the time needed to solve linear equation systems (Section 3.1.1). The computation time is comparable to the OpenMP version of the program. Note that the MPI version does not slow down considerably when utilizing more than one computation node.
That the MPI version scales very well can also be seen in Figure 6.4, where the amount of computation needed was increased with the number of CPUs utilized. Here the MPI version performs just as good as the OpenMP version. But the main benefit of the translation of MPI - that it can be used with more than one node - can be utilized very well.
The developed translation works very good in this case. Although more memory was used, as shown in Figure 6.5. Basically the MPI version requires double the amount of memory the OpenMP version uses, mainly attributable to the distribution of the data. This will be further discussed in Section 8.2. In contrast to a PDE, discussed in Section 6.6, there are no halo lines in this case. For each step only one line needs to be shared among all other processes (see Appendix A.1). Therefore the additional memory overhead does not grow as fast as in the case discussed in Section 6.6[1].

[1]In addition more data was distributed in this case, so the proportion of this additional overhead is smaller.

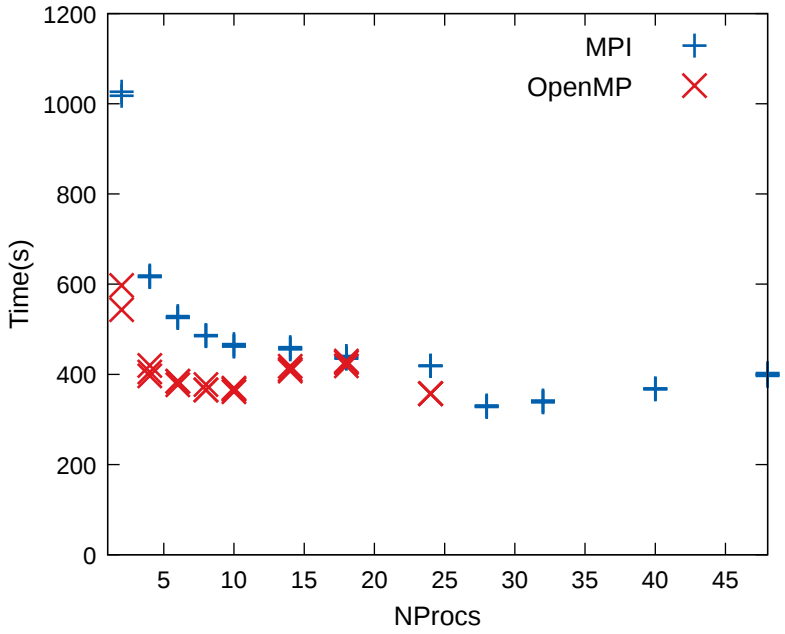

Figure 6.3.: Linear equation: Strong scaling.

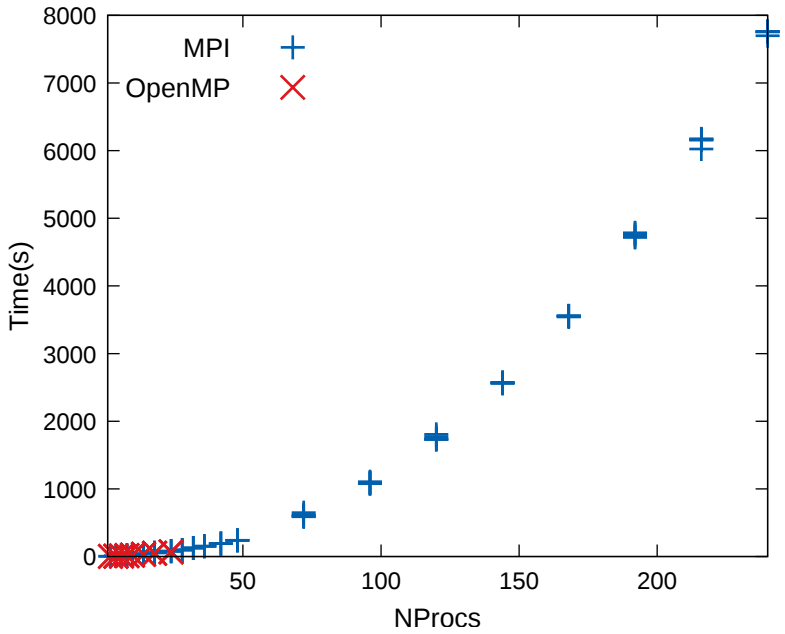

Figure 6.4.: Linear equation: Weak scaling. Number of variables used: $\sqrt{2 \cdot 10^6 \cdot n}$ (n is the number of CPU cores utilized). As Gaussian elimination is element of $O(n^3)$, a quadratic increase of the calculation time is expected in this case.

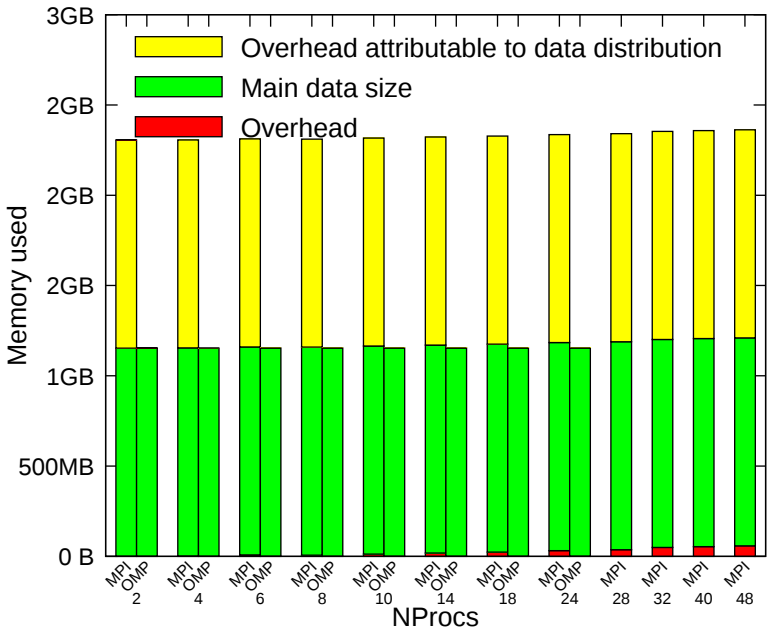

Figure 6.5.: Linear equation: Peak memory usage.

6.5. Fast Fourier transform

As one can see in Figure 6.6 the translation of the FFT-algorithm (Section 3.3.1) to MPI does not work quite well. The main problem is, that every step does require the copying of data in the MPI version, as the data needs to be transferred from one process to another. This is especially expensive when using more than one node. As the FFT-algorithm uses a butterfly communication scheme (Figure 3.3), there is no data locality that may be exploited, in order to lower the need for communication, as each datapoint will basically be combined with every other point. This means that regardless of the distribution of the data among the different processes, each process will need the data from every other process during the FFT-algorithm. This is shown in Figure 6.6 through the comparision of the MPI version using a distributed array (Section 5.6.2) and a master based array (Section 5.6.3). Although the distributed array works better when executing the MPI program on one node, they perform worse when utilizing more than one node, mainly attributable to the additional overhead needed to keep track which rank owns which array elements.

As all the data is needed in each processes memory space, the amount of memory needed grows significantly with every new process. This is shown in Figure 6.7. Therefore an analysis of the weak scaling behavior is left out in this thesis.

This leads to the conclusion that a translation to MPI is not worth the effort when using an all to all communication scheme, like a butterfly.

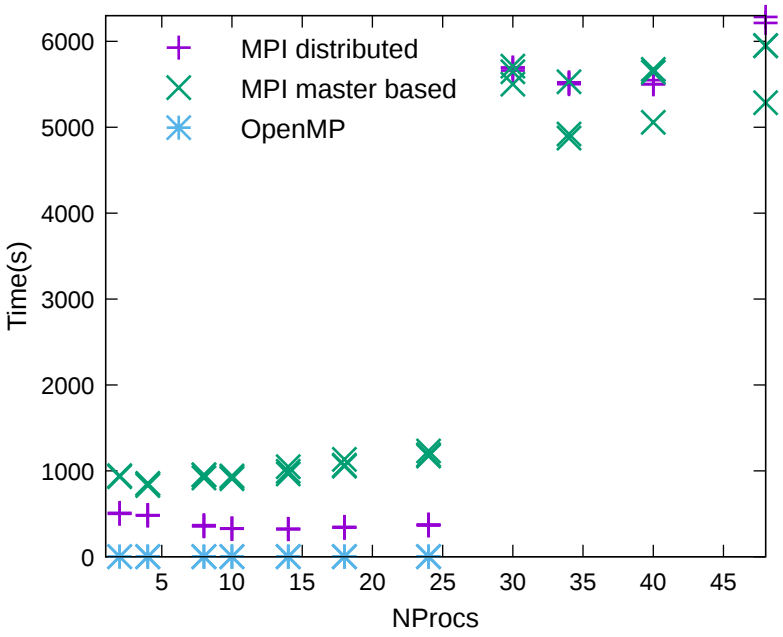

Figure 6.6.: Fast Fourier transform: Strong scaling.

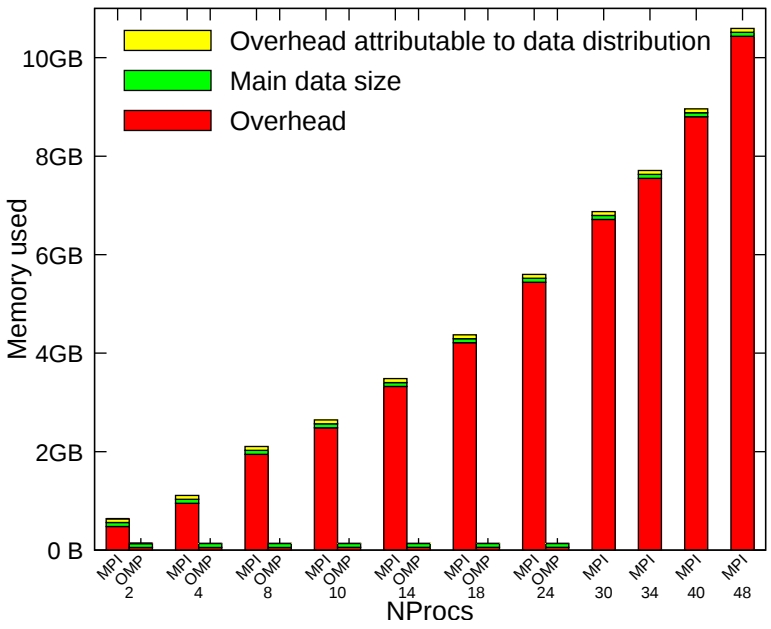

Figure 6.7.: Fast Fourier transform: Peak memory usage.

6.6. Partial differential equation

In Figure 6.8 one can see that translating a program to solve a partial differential equation (PDE) with a stencil code (Section 3.5.1) to MPI scales quite well. Although the MPI version is slower than the OpenMP version, when executing it with the same amount of CPUs, the benefit of MPI that it may be used with more than one node can be exploited very well. One can see that the runtime of the MPI version slightly decreases with more processes, despite the fact that inter-node communication is way more expensive than intra-node communication. The comparatively long runtime when using the same amount of CPUs is attributable to the fact that every communication in MPI has the need for copying data from the address space of one process to the space of another, while OpenMP threads may just read the data without the need of copying it.

The key advantage of MPI, namely that more processes are available, can be exploited quite reasonably by the presented automatic translation in this case. This can be seen in Figure 6.10 as well, where the problem size was increased with the amounts of CPUs. The effect that more utilized CPUs leads to more communication and synchronization overhead, which can also be seen in the OpenMP version, is stronger in the MPI version. Nevertheless the MPI version can utilize more CPUs to solve larger problems.

If the calculation should stop when a specific precision is reached, the MPI version does not scale that well when distributed among two nodes, as shown in Figure 6.10. This is mainly because the current precision has to be reduced after every iteration to see if the calculation stops. This reduction will take considerably longer when spread among more than one node.

The memory overhead incurred is shown in Figure 6.11. The overhead rises with more processes involved. The significant downside that a buffer for the data has to be allocated twice during the data distribution will be discussed in Section 8.2 further. Apart from that the overhead is quite reasonable considering it is spread among the different processes. One main source of the memory overhead are the halo lines, that need to be shared between two processes. Therefore the halo lines will be present on each of the neighbouring processes and will need double the amount of memory as they have their own buffer in each processes address space. As the number of halo lines increase with the number of processes, the overall overhead increases as well.

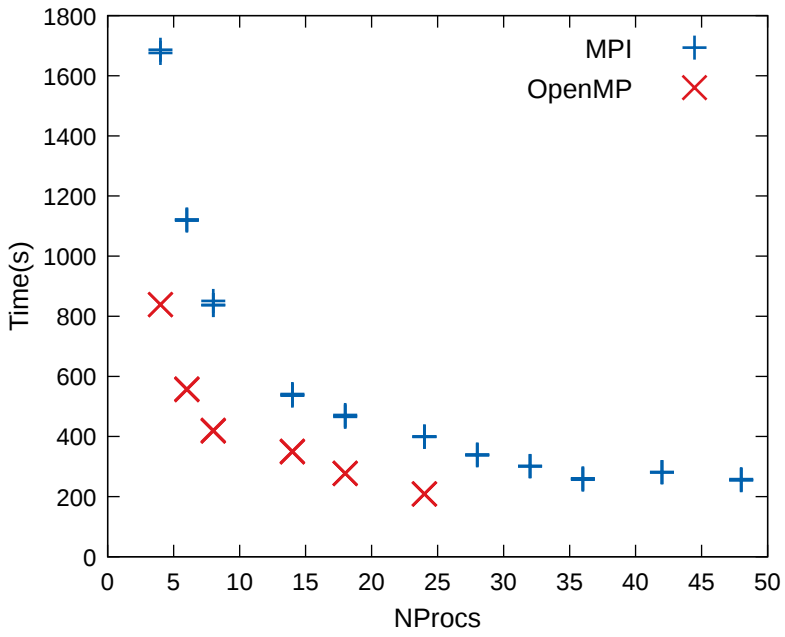

Figure 6.8.: PDE: Strong scaling. Termination from iteration number.

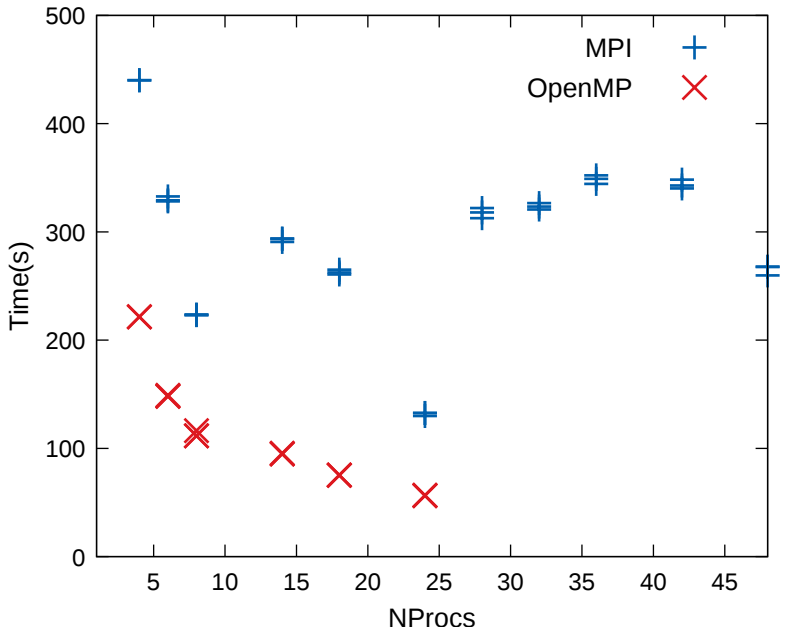

Figure 6.9.: PDE: Strong scaling. Termination when specific precision is reached.

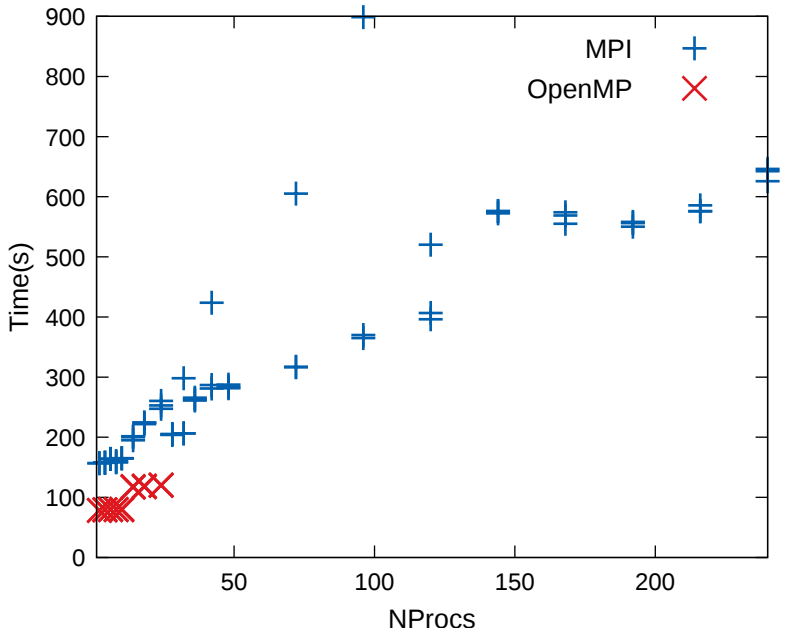

Figure 6.10.: PDE: Weak scaling (termination from iteration number). Number of matrix lines ($N \times N$ matrix) used : $9 \cdot \sqrt{3.75 \cdot 10^4 \cdot n}$ (n is the number of CPU cores utilized).

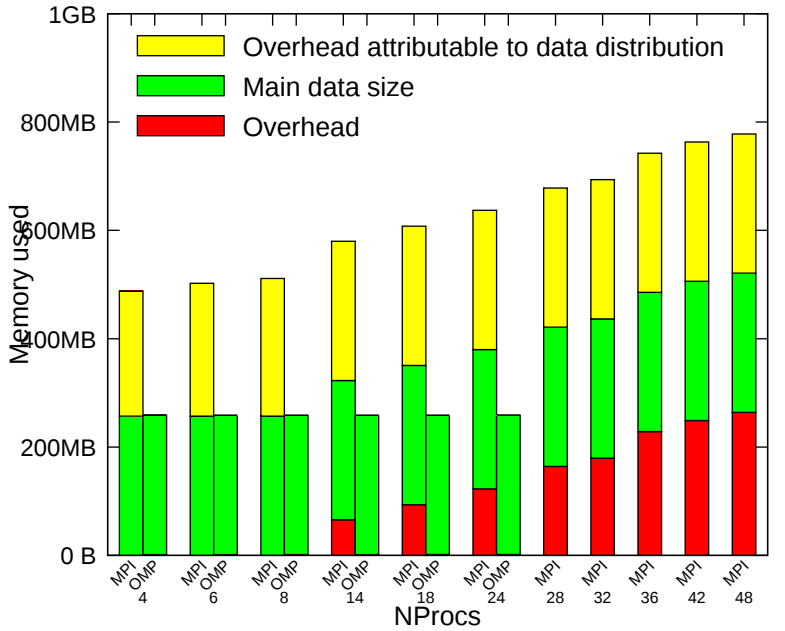

Figure 6.11.: PDE: Peak memory usage.

6.7. K-Means

The time needed to execute the K-Means algorithm (Section 3.7.1) on random datapoints is shown in Figure 6.12. As the datapoints were random, the number of times the cluster centers need to be refined (number of iterations) is also random. In general the MPI version does perform a bit worse than the OpenMP version, when executing it on one node. Having said that, the extended cost of communication when using more than one node plays an important role. Therefore, the reduction step to obtain the refined cluster centers takes considerably longer. In this test this fact was not compensated by the better performance of the map phase.

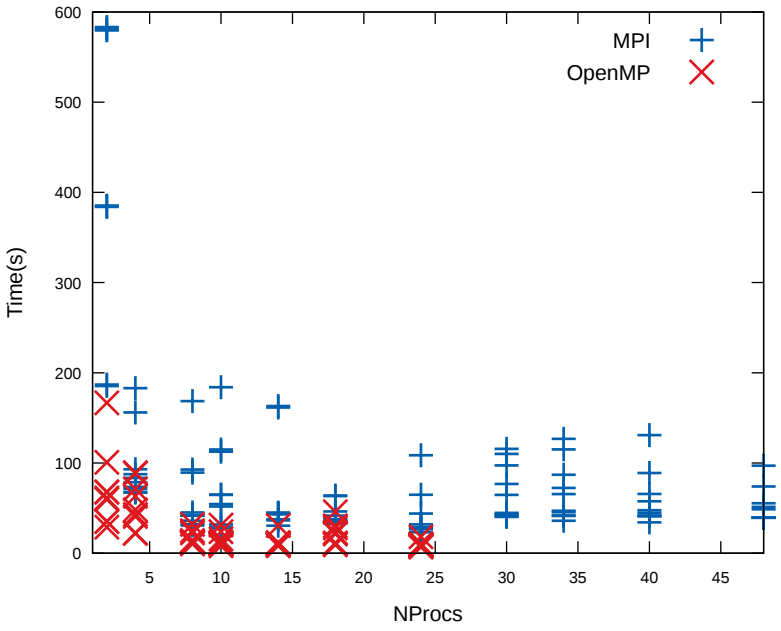

Figure 6.12.: K-Means algorithm: Strong scaling.

In this case the memory used by the MPI version is considerably higher than the memory used by the OpenMP version, as shown in Figure 6.13. This is attributable to the fact that the reduction is performed by the master process only (see Appendix A.4). Therefore each process will have a part of an array where it writes the local best cluster centers. The overhead involved with also distributing this array is quite high compared to the data size. The problem that twice the amount of memory needed during the initial distribution is also relevant for this additional array, which is one of the main reasons why the overhead is so large, compared to the data size. As this additional overhead greatly increases with more processes, a sophisticated analysis of the weak scaling behavior is not possible with the current version of the translation.

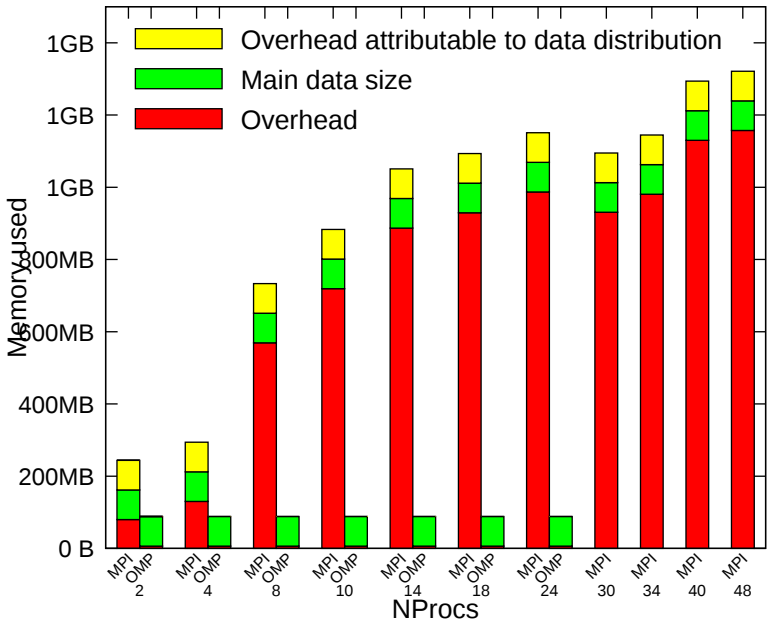

Figure 6.13.: K-Means algorithm: Peak memory usage.

6.8. Dynamic programming

As Section 6.10 will show that the scheduling overhead when using tasks is quite a significant factor, the implementation of the Smith-Waterman algorithm (Section 3.10), that is translated to MPI, does not use tasks (see Appendix A.5).

The runtime of the program executing the Smith-Waterman algorithm is shown in Figure 6.14. Key property to note is that the runtime increases significantly when utilizing more than one node. Also the worse performance when comparing it on one node should not be neglected. In contrast to a PDE, discussed in Section 6.6, more communication of the overlapping regions has to take place in regard to the amount of computation. The workload was distributed among the different processes as illustrated in Figure 3.6. After each step of the pipeline, a whole line from the other processes will become invalidated. This will lead to unnecessary communication as a great proportion of an array line will not have been changed.

The memory usage is shown in Figure 6.15, where one can see that the additional overhead is relatively small compared to the size of the data[2]. As for the other cases, the effect that twice the amount of memory is needed for the distribution of the data will be discussed in Section 8.2. As the current translation does not allow to allocate more memory than available on one node, the analysis of weak scaling behavior is spared out in this case (also refer to Section 8.2 for a discussion about this memory limit).

[2]Here the size of the data refers to the alignment score matrix and the input sequences.

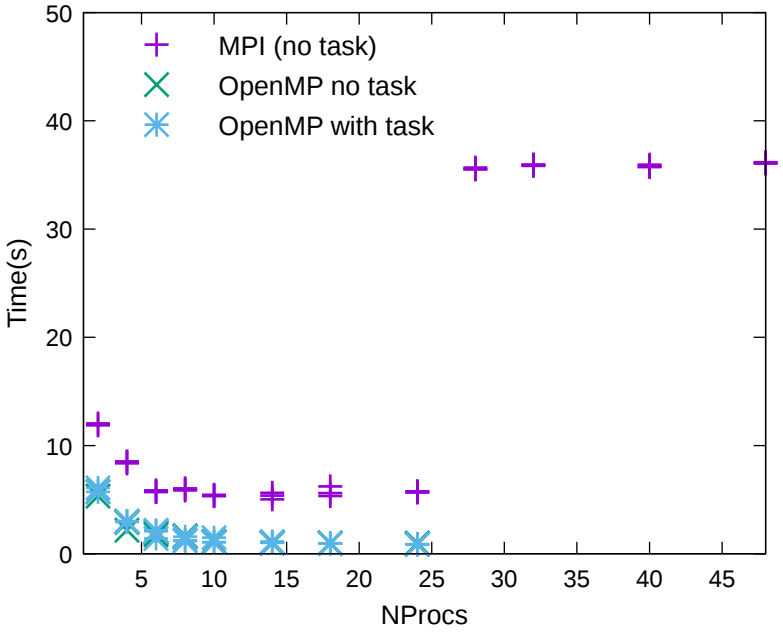

Figure 6.14.: Sequence alignment: Strong scaling.

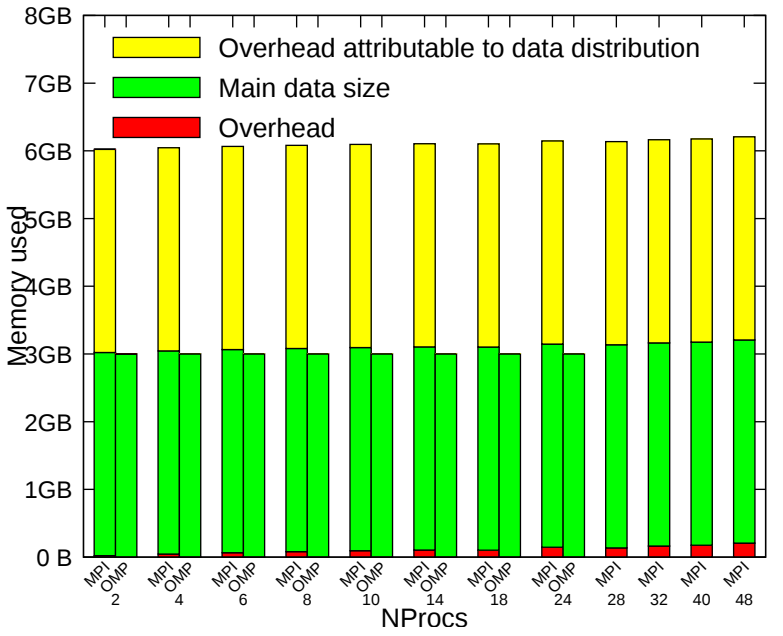

Figure 6.15.: Sequence alignment: Peak memory usage.

Overall, this shows that it is possible to translate a parallel dynamic programming algorithm to MPI. But the current version of the translation adds more communication than necessary and therefore can not yet exploit the full capabilities of multiple nodes in this case.

6.9. Branch and Bound

Figure 6.16 shows the time to solve the TSP problem (Section 3.11.1) for random points on a plain. One can see that the variance of the runtime is higher in the MPI version. This is explained in the higher cost of updating the currently found best solution in the MPI version. Therefore the runtime highly depends on the amount of times a better solution will be explored. Although it is significantly more expensive to communicate between two nodes, the MPI version does not considerably slow down, when using more than one node. That more processes can indeed explore the solution space faster does outweigh the fact that the communication is more expensive.

Overall this results look quite promising. Further tuning of the application to eliminate some communication might lead to a comparable execution time to the OpenMP version, but allowing for execution on a very large set of processors. For example one can only update the globally best solution found from time to time, in order to save communication.

As shown in Figure 6.17 the memory used by the MPI program is significantly larger than the memory used by the OpenMP program. Besides the overhead incurred when using processes instead of threads, this is mainly attributable to the fact that the adjacency matrix of the graph is duplicated to each process. OpenMP uses roughly 2.2 MB of memory. The memory used by the stacks is not significant (up to 1696 Bytes), whereas MPI uses on average roughly 3.6 MB per process (with a similar stack size). This additional memory consumption comes from all the communication information that needs to be stored. For example the mutex implementation uses a status buffer, that is shared through RMA, to indicate which processes are waiting for the mutex (refer Section 5.4.2).

For this use case I will spare out an analysis of the weak scaling behaviour as the amount of computation will increase factorially ($O(n!)$) in regard to the problem size. Therefore it is not possible to choose appropriate input parameters for testing, so that the amount of computation needed to solve the problem will increase with more CPUs used, while there is still a reasonable execution time for a large number of processes.

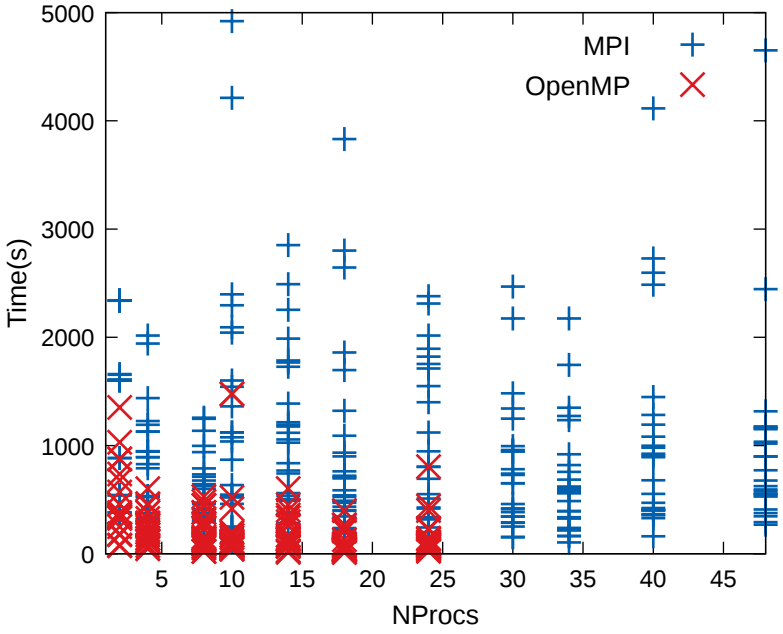

Figure 6.16.: TSP: Strong scaling.

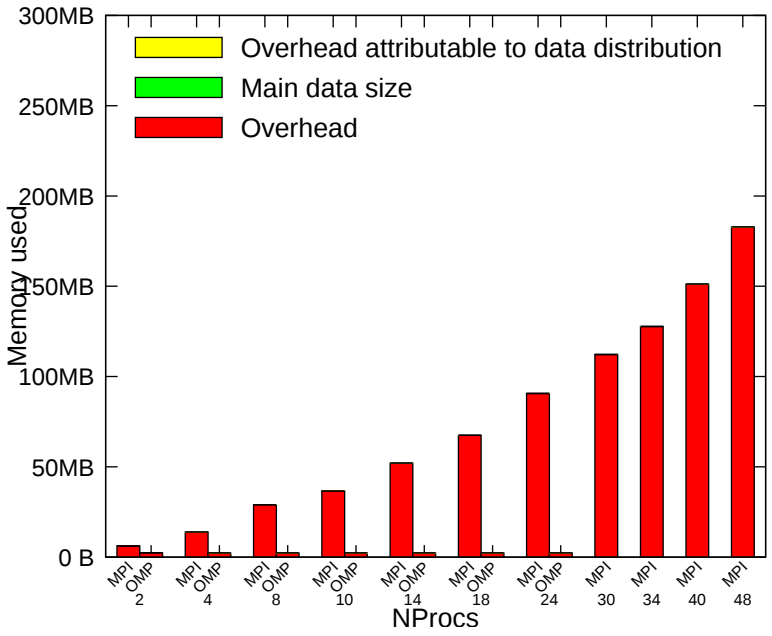

Figure 6.17.: TSP: Peak memory usage.

6.10. Task pragma

This section evaluates the translation of the OpenMP task pragma to MPI.

The used test program performs a simple strictly local computation on an 2-D array. In one version the OpenMP task pragma was used in order to create a task for each array line. In Figure 6.18 this is compared to a version, where the same work is distributed using the OpenMP for pragma. The example program used is described in Appendix A.7.

Most important property to note is, that the version of the test program that uses tasks, performs considerably worse than the version that uses the for pragma. The task scheduling overhead is already noticeable in the OpenMP version (see Figure 6.18). In this case, the weaker performance of the worksharing with the use of the task pragma is strongly intensified by the translation to MPI. In Figure 6.18 only the time needed when utilizing two processes is shown (top left). With more processes the task scheduling took way longer, up to 3500 seconds when utilizing 48 processes spread among two nodes. First, scheduling needs synchronization, which is more costly in MPI. Second, the scheduling strategy used (see Section 5.5.4) does not account for data locality. This means that a task may have to be executed on a rank which does not own (see Section 5.6.2) the involved array line associated with the task, which will lead to further communication overhead.

Therefore the translation of the task pragma in its current state is a *proof of concept* for the translation to MPI, but much more effort in regards to scheduling is required (see Section 8.2).

The memory consumption of the version of the test program that uses tasks instead of for pragma for worksharing is shown in Figure 6.19. One can see that in this case, the usage of the task pragma leads to a significant memory overhead in the OpenMP version (compared to the data size). For the MPI version one key property to note is, that the memory usage is not deterministic in this case. The total amount of memory needed is dependent on the amount of local buffer that is used for each process in order to work on not owned array lines. As discussed above, with more processes, the probability of working on a task that does not use the own array lines is higher, therefore it is likely that more memory is needed.

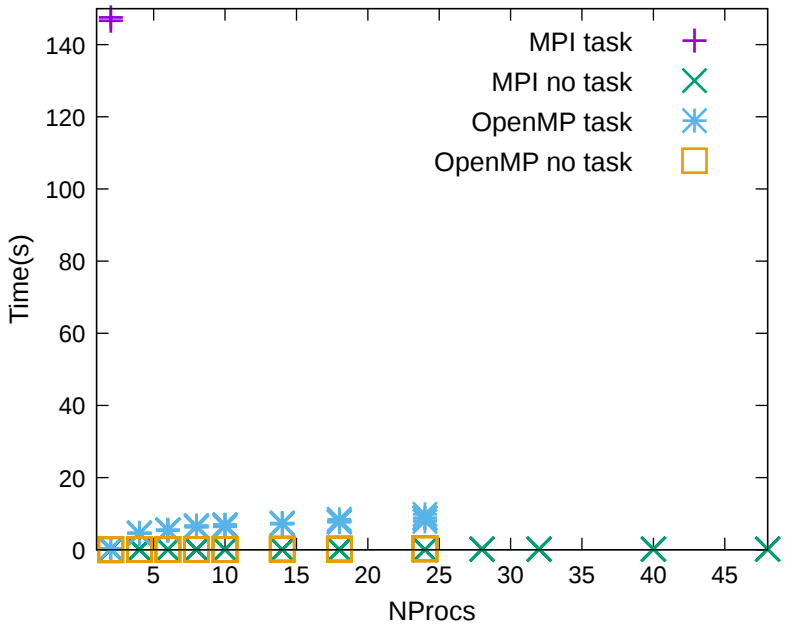

Figure 6.18.: Comparison of a program using OpenMP task and for pragma for workshare.

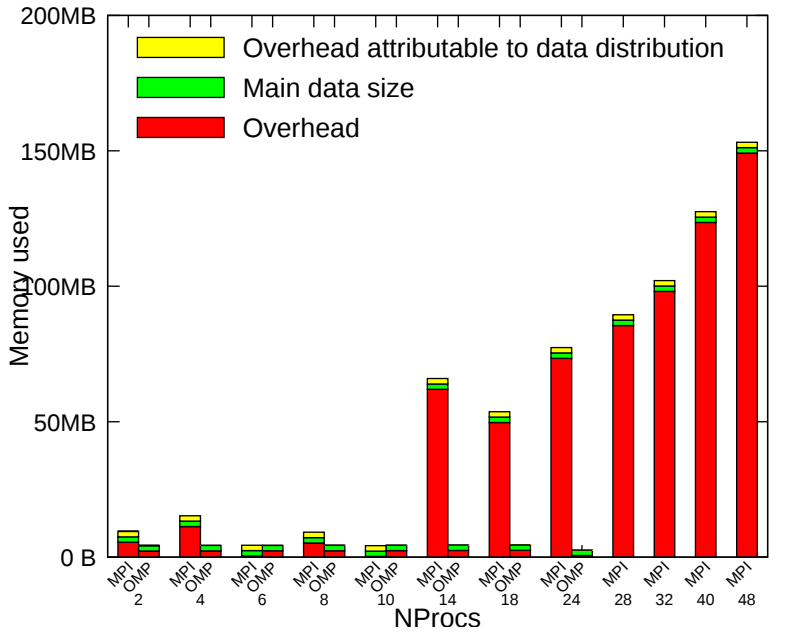

Figure 6.19.: Memory consumption of a program using the task pragma for workshare.

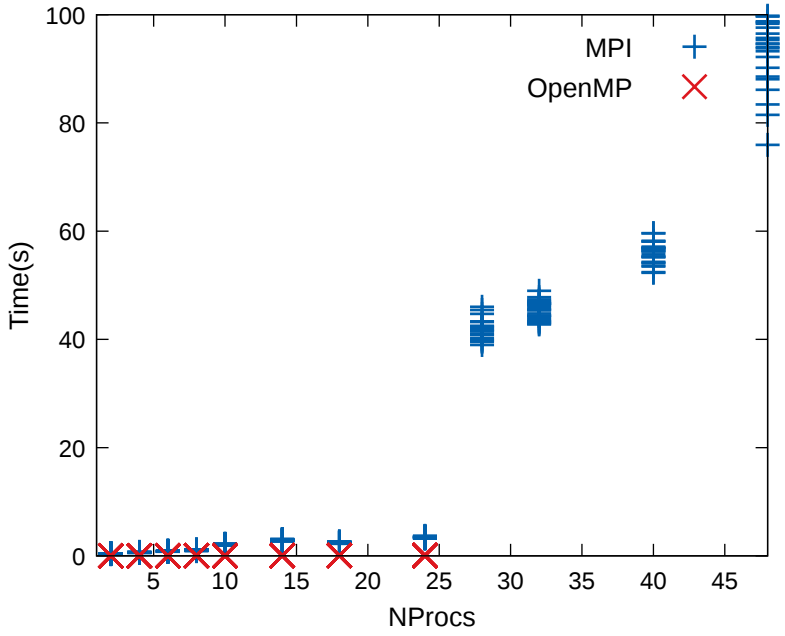

Figure 6.20.: Searching random elements in distributed search tree.

6.11. Distributed search tree

In order to evaluate the performance of the translation of shared struct pointers (see Section 5.8), I used a binary search tree.
The time needed to search for 1000 elements per thread in a binary search tree is shown in Figure 6.20. The tree was initialized with every thread inserting 1000/numThreads random elements. Therefore the tree is not necessarily balanced but as the root element is the middle of the range from which the random elements came, a fairly balanced tree is very likely. The time needed to insert those elements was not measured.

One can clearly see that there is a huge difference when scaling up the distributed search tree to more than one computation node. This is explained in the latency of the communication, which is greatly increased when communicating between different computation nodes. One can also see in Figure 6.20 that the variance increases when using more than one computation node. If a process must traverse the tree and finds a tree node, that is stored on the other computation node, communication has to take place in order to access this tree node. Every time this "remote" tree node has to be visited communication has to take place. This is necessary because this particular tree node might be updated by another process between two visits[3]. Therefore the more distributed the processes are among the computation nodes, the more likely it gets that a visited node resides on the other computation node.

For this particular case where only searching within a static tree was measured, it

[3]Although an update does not happen in this test program in order to avoid race conditions.

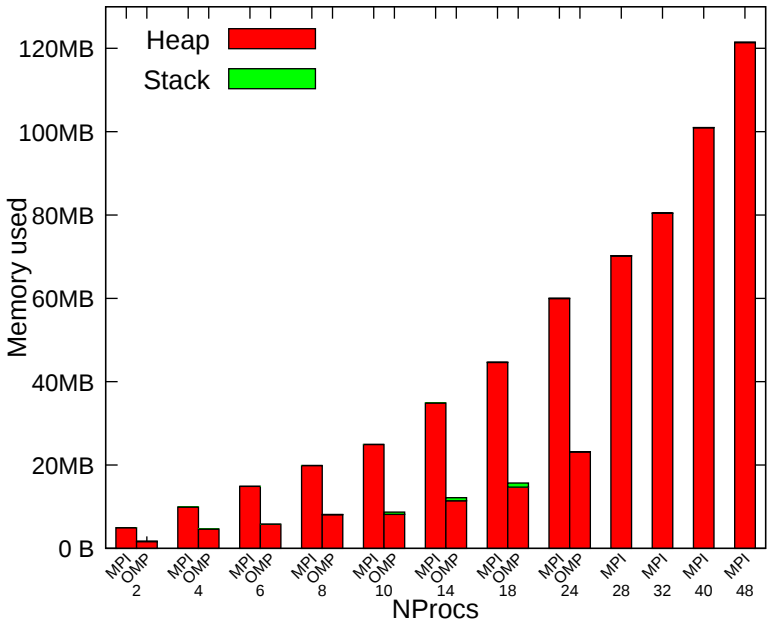

Figure 6.21.: Peak memory usage of the distributed search tree.

will be better to duplicate the tree, as the communication within the node can be done through shared memory just similar to OpenMP. This allows for less communication when searching within the tree is the most common task. But this comes with a downside, as rearrangement of the tree is much more costly, because it then has to be applied to every copy of the tree. This might also lead to more synchronization overhead in order to avoid race conditions as an update to the tree will take longer.

But the duplication of the tree will result in an even larger memory usage. Figure 6.21 shows the memory consumed. The memory consumption of the MPI version is significantly higher than the memory usage of the OpenMP version. Note that the exact amount of memory needed is dependent on the tree's depth and therefore not deterministic in this case, as the tree is visited with a recursive function.
On average, the OpenMP version needs about 0.9 MB per thread, while the MPI version uses about 2.5 MB per process. The increased need of memory is mainly attributable to the extra information needed, when sharing pointers to shared structs (refer Section 5.8).

Therefore I do not advise to spread such dynamic linked datastructures like trees among different nodes if possible. Nevertheless this thesis showed - as a *proof of concept* - that it is possible to also translate this kind of (possibly dynamic) complex structures to MPI.

7. Related work

One approach for using OpenMP on distributed memory systems is based upon software distributed shared memory systems (e.g. [HLCZ00, SHHI01, KKH03, CCM+04, MI03, KLB02]). But these systems often incur a large overhead especially at synchronization points [BME02]. Additionally many systems impose a limit on the shared memory that may be allocated [HCL05]. This prevents their application to large scale problems.

[LSB07] proposes OpenMPD, which "provides OpenMP-like directives to describe array distribution and work sharing on the loop on parallel processes for data parallel programming" [LSB07]. But this project did not go beyond the state of a first prototype, as implementing and learning a new technology is too time consuming for the users. Also this thesis aims at scaling up existing OpenMP applications, in order to avoid the need of porting them to a different technology.

[Ren07] uses checkpointing techniques to introduce parallel computing inside sequential programs. Basically a checkpoint is written before a parallel region. After the parallel region the resulting checkpoints from all processes are merged in order to get the resulting state. This approach called Checkpointing Aided Parallel Execution (CAPE) is then used to distribute OpenMP code. With this approach, [Ren07] developed a distributed OpenMP compiler on top of GCC. However the main disadvantage of this approach is that no shared variables are allowed in parallel sections, as it is necessary, that all parallel parts satisfy Bernstein's conditions [Ber66].

In some aspects the approach of [KJM+11] is comparable to the work presented in this thesis. [KJM+11] uses Cetus [DBM+09] to translate an OpenMP to a MPI program. One major disadvantage, compared to this work, is that [KJM+11] does not use MPI one sided communication, therefore the full capability of low latency interconnect hardware may not be exploited by this approach. Furthermore this thesis also includes a *proof of concept* for translating linked datastructures and the OpenMP task pragma directive. Compared to Cetus the LLVM compiler infrastructure used in this thesis is performing better at compilation time and the IR is closer to machine code, which allows to relate it more closely to the instructions that ultimately will be executed [RMG+13]. Additionally Cetus only supports C whereas LLVM also offers frontends for other languages like Fortran.

8. Conclusion

8.1. Summary

This thesis explored the possibility of translating OpenMP code to MPI using the LLVM infrastructure. Therefore a LLVM pass was implemented, that translates OpenMP to MPI (Chapter 5).

The translation works correct in all tested cases, although not all features of OpenMP are implemented yet and some additional restrictions apply. For example the `omp atomic` pragma is not supported yet and it is currently not allowed to change the size of a shared array. But the most important OpenMP pragma directives can be used, which allows for the translation of many OpenMP programs. The translation of the `task` pragma and dynamic linked datastructures such as search trees is included as a proof of concept.

The performance of the translated programs depends upon the analyzed case.
Regular communication patterns such as a stencil code (Section 6.6) or Gaussian elemination (Section 6.4) can strongly benefit from a translation to MPI, as the better scalability of MPI can be exploited quite well in this cases.
For other cases, such as Branch and Bound (Section 6.9) and dynamic programming (Section 6.8) techniques as well as map-reduce alike tasks (Section 6.7), the performance of the MPI version still looks promising, although further tuning is needed.
For computation that uses an all-to-all communication scheme, such as a fast fourier transform (Section 6.5), the currently implemented translation to MPI does not seem beneficial.

All in all, the results of this thesis are very promising. During the implementation of the LLVM pass that translates OpenMP to MPI a special focus was placed on the possibility to further extend its capabilities in the future (also refer Appendix B), therefore the implemented pass will be a good foundation for future research.

8.2. Future work

As explained above, the implemented pass will provide a solid basis for future research. Some interesting points for further investigation include:

- Distributed reading of (input) data:
  As discovered in Chapter 6, during the initial distribution of the data, twice as much memory is needed. Additionally the memory allocation within the serial

region is limited to the size of the memory of a single node. In order to further scale up and fully exploit the capabilities of more computation nodes, it is desirable to distribute the data already in the sequential region of the program. For large sets of input data, that are read from disk, it might be worth exploring the possibility of parallel I/O so that the data is initialized distributed. This might also eliminate the communication overhead associated with the initial data distribution.

- Advanced scheduling:
  Except for the *proof of concept* for the translation of the task pragma (Section 5.5.4), within this thesis only static schedules were considered. OpenMP also offers dynamic scheduling of for loop iterations to improve load balancing. Therefore exploring the possibilities of transferring dynamic schedules to the MPI version is an interesting field. A special focus may be placed on scheduling strategies that consider data locality, in order to avoid as much communication as possible (also see Section 6.10).

- Support the programmer to choose an appropriate communication pattern:
  Currently the appropriate communication pattern for each variable has to be specified by the programmer using the introduced `#pragma omp2mpi comm` directive (see Chapter 4). Another interesting topic is, if a good communication pattern may also be found automatically, for example by trying different communication patterns in a small test run of the application.

- Further enhancing of implemented or development of new communication patterns:
  Of course further enhancing the communication patterns used by the translated program is an interesting task. E.g. Section 6.8 points out a case where unnecessary communication takes place for the current implementation of distributed arrays. Additionally there is room for more communication patterns e.g. all-to-all communication schemes like a butterfly (Section 6.5). As the test system is connected with Gigabit-Ethernet (refer Section 6.1), evaluating the performance on a system with interconnection hardware designed to use with RMA operations is an important task, as the current implementation of the communication uses one sided MPI operations.

- Only use MPI for inter node communication:
  Hybrid programs that use MPI for inter-node and OpenMP for intra-node parallelism are often used in the HPC environment, as they often perform better than pure MPI programs [BM08]. Therefore it is an interesting work to see if an OpenMP program also can be translated to a hybrid program that still uses threads for intra node parallelism in order to achieve a better performance.

Bibliography

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[ABD+09] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[Akl89] Selim G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice Hall, 1989.

[And14a] Thomas Andreae. Mathematik II für Studierende der Informatik (Analysis und Lineare Algebra). 2014.

[And14b] Thomas Andreae. Optimierung für studierende der informatik. 2014.

[Bar14a] Brandon Barker. Cornell virtual workshop on mpi collective communications. `https://cvw.cac.cornell.edu/MPIcc/` [as of 12.7.2018], 2014.

[Bar14b] Brandon Barker. Cornell virtual workshop on mpi point-to-point communication. `https://cvw.cac.cornell.edu/MPIP2P/` [as of 11.7.2018], 2014.

[Bar17] Brandon Barker. Cornell virtual workshop on openmp. `https://cvw.cac.cornell.edu/OpenMP/` [as of 23.7.2018], 2017.

[Ber66] Arthur J Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, (5):757–763, 1966.

[Ble17] Michael Blesel. Compiler assisted translation of OpenMP to MPI using LLVM. Online `https://wr.informatik.uni-hamburg.de/_media/research:theses:michael_blesel_compiler_assisted_translation_of_openmp_to_mpi_using_llvm.pdf`, 10 2017.

[BM08] Holger Brunst and Bernd Mohr. Performance analysis of large-scale openmp and hybrid mpi/openmp applications with vampir ng. In *OpenMP Shared Memory Parallel Programming*, pages 5–14. Springer, 2008.

[BME02] Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann. Towards openmp execution on software distributed shared memory systems. In *International Symposium on High Performance Computing*, pages 457–468. Springer, 2002.

[Boa17] OpenMP Architecture Review Board. Openmp faq. `https://www.openmp.org/about/openmp-faq/` [as of 23.7.2018], 2017.

[Bos04] Marina Bosi. Introduction to digital audio coding and standards. *Journal of Electronic Imaging*, 13(2):399, apr 2004.

[CAH+11] Robert D Cameron, Ehsan Amiri, Kenneth S Herdy, Dan Lin, Thomas C Shermer, and Fred P Popowich. Parallel scanning with bitstream addition: An xml case study. In *European Conference on Parallel Processing*, pages 2–13. Springer, 2011.

[Can87] John Canny. A computational approach to edge detection. In *Readings in Computer Vision*, pages 184–203. Elsevier, 1987.

[CCM+04] Juan Jose Costa, Toni Cortes, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Running openmp applications efficiently on an everything-shared sdsm. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 35. IEEE, 2004.

[CHQ+12] Claudio Canuto, M Yousuff Hussaini, Alfio Quarteroni, A Thomas Jr, et al. *Spectral methods in fluid dynamics*. Springer Science & Business Media, 2012.

[CJ12] Heinz Konietzky Christian Jakob. Partikelmethoden - eine Übersicht -. `https://tu-freiberg.de/fakult3/gt/feme/studium/Handbuch_Partikelmethoden.pdf` [as of 24.05.2018], 2012.

[Cla99] Jens Clausen. Branch and bound algorithms-principles and examples. *Department of Computer Science, University of Copenhagen*, pages 1–30, 1999.

[cla18a] Clang 8 documentation - assembling a complete toolchain. `https://clang.llvm.org/docs/Toolchain.html#tools` [as of 13.9.2018], 2018.

[cla18b] Clang 8 documentation - clang plugins. `https://clang.llvm.org/docs/ClangPlugins.html` [as of 13.9.2018], 2018.

[cla18c] clang::pragmanamespace class reference. `https://clang.llvm.org/doxygen/classclang_1_1PragmaNamespace.html` [as of 13.9.2018], 2018.

[CM99] Peter Carr and Dilip Madan. Option valuation using the fast fourier transform. *The Journal of Computational Finance*, 2(4):61–73, 1999.

[Col04] Phillip Colella. Defining software requirements for scientific computing, 2004.

[Coo15] William J. Cook. *In Pursuit of the Traveling Salesman*. Princeton University Press, jan 2015.

[Cor18] NVIDIA Corporation. Cuda llvm compiler. `https://developer.nvidia.com/cuda-llvm-compiler` [as of 19.10.2018], 2018.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–297, may 1965.

[Dan63] George B Dantzig. Linear programming and extensions. princeton landmarks in mathematics and physics, 1963.

[DBL17] *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2017, Denver, CO, USA, November 13, 2017*. ACM, 2017.

[DBM+09] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12), 2009.

[DL42] G.C. Danielson and C. Lanczos. Some improvements in practical fourier analysis and their application to x-ray scattering from liquids. *Journal of the Franklin Institute*, 233(4):365–380, apr 1942.

[Fin] Ian Finlayson. Collective communication - mpi_allreduce. `http://ianfinlayson.net/class/cpsc425/notes/images/all-reduce2.png` [as of 1.11.2018].

[For15] Message Passing Interface Forum. Mpi: A message-passing interface standard version 3.1. `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf` [as of 11.7.2018], 2015.

[get] gettimeofday(2) - linux man page. `http://man7.org/linux/man-pages/man2/gettimeofday.2.html` [as of 24.09.2018].

[HCL05] Lei Huang, Barbara Chapman, and Zhenying Liu. Towards a more efficient implementation of openmp for clusters via translation to global arrays. *Parallel Computing*, 31(10-12):1114–1139, 2005.

[HH79] Edwin Hewitt and Robert E. Hewitt. The gibbs-wilbraham phenomenon: An episode in fourier analysis. *Archive for History of Exact Sciences*, 21(2):129–160, 1979.

[HLCZ00] Y Charlie Hu, Honghui Lu, Alan L Cox, and Willy Zwaenepoel. Openmp for networks of smps. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, 2000.

[IC07] The ISO/IEC 9899 Committee. *The ISO C99 Standard (ISO/IEC 9899:TC3 Committee Draft)*. 09 2007.

[Ind] RAMALINGAM R (Samsung India). Performance on android & openmp. `https://pdfs.semanticscholar.org/presentation/80b7/865eb6c2b546b26b674820a20e3111d26a2e.pdf` [as of 05.06.18].

[KJM+11] Okwan Kwon, Fahed Jubair, Seung-Jai Min, Hansang Bae, Rudolf Eigenmann, and Samuel P Midkiff. Automatic scaling of openmp beyond shared memory. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 1–15. Springer, 2011.

[KKH03] Yang-Suk Kee, Jin-Soo Kim, and Soonhoi Ha. Parade: An openmp programming environment for smp cluster systems. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 6–6. IEEE, 2003.

[KLB02] Sven Karlsson, Sung-Woo Lee, and Mats Brorsson. A fully compliant openmp implementation on software distributed shared memory. In *International Conference on High-Performance Computing*, pages 195–206. Springer, 2002.

[KT06] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

[Kun16] Julian M. Kunkel. Map reduce & hadoop - lecture bigdata analytics. `https://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2016_2017/bd-04.pdf` [as of 24.05.2018], 2016.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[LB14] Steve Lantz and Brandon Barker. Cornell virtual workshop on mpi one-sided communication. `https://cvw.cac.cornell.edu/MPIoneSided/` [as of 11.7.2018], 2014.

[lib] Gnu libgomp: Implementing reduction clause. `https://gcc.gnu.org/onlinedocs/gcc-8.1.0/libgomp/Implementing-REDUCTION-clause.html#Implementing-REDUCTION-clause` [as of 24.05.2018].

[Llo82] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[llva] Llvm language reference manual. https://llvm.org/docs/LangRef.html [as of 22.6.2018].

[llvb] Llvm logo. http://llvm.org/Logo.html [as of 22.6.2018].

[llvc] The llvm project. the llvm compiler infrastructure. http://llvm.org/ [as of 22.6.2018].

[llvd] Llvm's analysis and transform passes. https://llvm.org/docs/Passes.html [as of 22.6.2018].

[llve] Openmp®: Support for the openmp language. http://openmp.llvm.org/ [as of 22.6.2018].

[LLV18a] LLVM. llvm::module class reference. http://llvm.org/doxygen/classllvm_1_1Module.html [as of 18.6.2018], 2018.

[LLV18b] LLVM. llvm::type class reference. http://llvm.org/doxygen/classllvm_1_1Type.html [as of 18.8.2018], 2018.

[LSB07] Jinpil Lee, Mitsuhisa Sato, and Taisuke Boku. Design and implementation of openmpd: An openmp-like programming language for distributed memory systems. In *International Workshop on OpenMP*, pages 143–147. Springer, 2007.

[LSEJ11] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. Speculative parallel pattern matching. *IEEE Transactions on Information Forensics and Security*, 6(2):438–451, 2011.

[Lud17] Thomas Ludwig. Vorlesung hochleistungsrechnen. https://files.wr.informatik.uni-hamburg.de/s/861a1cc4ebd9bea52e95e635dac45b94/download?path=%2F&files=hr-1718-teil2.pdf[as of 12.7.2018], 2017.

[LW66] Eugene L Lawler and David E Wood. Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719, 1966.

[mem18] Llvm's analysis and transform passes - mem2reg: Promote memory to register. https://llvm.org/docs/Passes.html#mem2reg-promote-memory-to-register [as of 19.10.2018], 2018.

[MI03] Hiroya Matsuba and Yutaka Ishikawa. Openmp on the fdsm software distributed shared memory. In *The Fifth European Workshop on OpenMP, EWOMP*, volume 3, 2003.

[MMS14] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 529–542. ACM, 2014.

[mpi] mpi_comm_spawn(3) - linux man page. https://linux.die.net/man/3/mpi_comm_spawn [as of 27.6.2018].

[MSM04] Timothy G. Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming (Software Patterns Series)*. Addison-Wesley Professional, 2004.

[OA10] Jorge Luis Ortega-Arjona. *Patterns for Parallel Software Design*. Wiley, 2010.

[Ope13] ARB OpenMP. Openmp application program interface version 4.0. https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf [as of 24.05.2018], 2013.

[Pap77] Christos H. Papadimitriou. The euclidean travelling salesman problem is NP-complete. *Theoretical Computer Science*, 4(3):237–244, jun 1977.

[PEASJ96] Bradford W Parkinson, Per Enge, Penina Axelrad, and James J Spilker Jr. *Global positioning system: Theory and applications, Volume II*. American Institute of Aeronautics and Astronautics, 1996.

[Pre11] Andreas Prell. Simple mutex implementation based on mpi-2 rma. https://gist.github.com/aprell/1486197 [as of 14.8.2018], 2011.

[Rab89] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[Ren07] Éric Renault. Distributed implementation of openmp based on checkpointing aided parallel execution. In *International Workshop on OpenMP*, pages 195–206. Springer, 2007.

[RMcKB97] Gerald Roth, John Mellor-crummey, Ken Kennedy, and R. Gregg Brickner. Compiling stencils in high performance fortran. In *In Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, pages 1–20. ACM Press, 1997.

[RMG+13] Gabriel Rodríguez, María J Martín, Patricia González, Juan Touriño, and Ramón Doallo. Compiler-assisted checkpointing of parallel codes: The cetus and llvm experience. *International Journal of Parallel Programming*, 41(6):782–805, 2013.

[Rom04] P. Romero. Bioinformatics: Sequence and genome analysis. *Briefings in Bioinformatics*, 5(4):393–396, jan 2004.

[RWZ88] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM, 1988.

[SHHI01] Mitsuhisa Sato, Hiroshi Harada, Atsushi Hasegawa, and Yutaka Ishikawa. Cluster-enabled openmp: An openmp compiler for the scash software distributed shared memory system. *Scientific Programming*, 9(2-3):123–130, 2001.

[SW81] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, mar 1981.

[Tim16] Matt Timmermans. In common tongue, what is the differences between sparse and dense matrices? `https://math.stackexchange.com/questions/1632739/in-common-tongue-what-is-the-differences-between-sparse-and-dense-matrices` [as of 20.10.2018], 2016.

[TM16] Cheng Tang and Claire Monteleoni. On lloyd's algorithm: New theoretical insights for clustering in practice. In *Artificial Intelligence and Statistics*, pages 1280–1289, 2016.

[Vah96] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Pearson, 1996.

[Val17] Valgrind Developers. Massif: a heap profiler. `http://valgrind.org/docs/manual/ms-manual.html` [as of 24.09.2018], 2017.

[VSGS13] Vikas, Travis Scott, Nasser Giacaman, and Oliver Sinnen. Using OpenMP under android. In *OpenMP in the Era of Low Power Devices and Accelerators*, pages 15–29. Springer Berlin Heidelberg, 2013.

[Wika] Wikipedia. Fast fourier transform. `https://en.wikipedia.org/wiki/Fast_Fourier_transform` [as of 16.8.2018].

[Wikb] Wikipedia. Smith–waterman algorithm. `https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm` [as of 16.8.2018].

[wit18a] Projects built with llvm. `https://llvm.org/ProjectsWithLLVM/` [as of 19.10.2018], 2018.

[Wit18b] Kim Wittenburg. Statische code-optimierung mit llvm und polly. `https://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2017_2018/ep-1718-wittenburg-polly_durckversion.pdf` [as of 22.06.2018], 2018.

[Wun01] Holger Wunsch. Der baum-welch algorithmus für hidden markov models, ein spezialfall des em-algorithmus. `http://www.sfs.uni-tuebingen.de/resources/em.pdf` [as of 24.7.1018], 2001.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[yK09] Øystein Krog. Particle simulations on the gpu. http://www.idi.ntnu.no/~elster/tdt24/tdt24-f09/krog-Particle-Sims-on-GPU.pdf [as of 20.10.2018], 2009.

Appendices

A. Listings of the used examples for the use cases

This chapter shows the main part of the techniques used as examples for the different use cases (refer to Chapter 3).

A.1. Linear equation system

```
1  // elemination and substitution in one step to exploit more
       ↪ parallelism
2  // n variables
3  // matrix therefore is n times n+1
4  // returns the solution vector; if infinite solutions: will not
       ↪ calculate solution
5  double* parallel_gaussian_elemination(int n, double** matrix) {
6      int i, j, k, p;
7      double factor;
8      for (k = 0; k < n; ++k) {
9          if (matrix[k][k] == 0) {            // pivot row if needed
10             for (p = k + 1; p < n; ++p) {
11                 if (matrix[p][k] != 0) {
12                     swap_rows(matrix, n + 1, k, p);
13                     break;
14                 }
15             }
16             if (p == n) { // no suitable pivot row was found
17                 printf("Not solvable or infinite solutions\n");
18                 //return nullptr;
19                 // in order to measure runtime of whole calculation:
                       ↪ pretend that it is solvable
20             }
21         }
22 #pragma omp parallel for default(none) shared(matrix,n,k)
       ↪ private(i,j,factor)
23         for (i = 0; i < n; ++i) {
24             if (i != k) {           // not eliminate it in its own row
25                 factor = matrix[i][k] / matrix[k][k];
26                 for (j = k + 1; j < n + 1; ++j) {
27                     matrix[i][j] = matrix[i][j] - factor *
                           ↪ matrix[k][j];
28                 }
```

```
29              }
30          }
31      }
32
33  // store the final value to solution:
34      double* solution = (double*) malloc(n * sizeof(double));
35      assert(solution != nullptr);
36  #pragma omp parallel for default(none) shared(matrix,n,solution)
        ↪ private(i)
37      for (i = 0; i < n; ++i) {
38          solution[i] = matrix[i][n]/matrix[i][i];
39      }
40      return solution;
41  }
```

Listing A.1: Solving a linar equation system with Gaussian Elemination method.

Listing A.1 shows the code to solve a linear equation system with the Gaussian elemination method. In order to exploit more parallelism, the elemination and substitution step is performed at the same time. This means that instead of a triangular matrix the result of the first step is a matrix where only the diagonal is nonzero. This is achived through applying the elemination to all rows (excluding self) (line 24 of Listing A.1) instead of applying it only to lower rows as in the traditional elimination step. Therefore the second substitution step of the Gaussian elemination algorithm is quite simple (line 38). The rearrangement of matrix rows (lines 10 to 22) is only done if this is needed, so that each thread may keep its own matrix rows.

For the matrix, the memory aware variant of distributed arrays (Section 5.6.2) was used.

A.2. Fast Fourier transform

```
1   // nn = number of data (power of 2)
2   // data = double[nn*2] (even indices: real part, odd indices:
        ↪ imaginary part)
3   void FFT(int nn, double data[]) {
4       unsigned long n, mmax, m, j, istep, i;
5       double wtemp, wr, wpr, wpi, wi, theta;
6       double tempr, tempi;
7
8   // reverse-binary reindexing
9       n = nn << 1;
10      j = 1;
11      for (i = 1; i < n; i += 2) {
12          if (j > i) {
13              SWAP(data[j - 1], data[i - 1]);
14              SWAP(data[j], data[i]);
15          }
16          m = nn;
```

```
17          while (m >= 2 && j > m) {
18              j -= m;
19              m >>= 1;
20          }
21          j += m;
22      }
23  // Danielson-Lanczos
24  #pragma omp parallel default(none)
        ↪ private(m,i,tempr,tempi,wr,wi,wpr,wpi,theta,wtemp,istep,mmax)
        ↪ shared(data,n)
25      {
26          mmax = 2;
27          while (n > mmax) {
28              istep = mmax << 1;
29              theta = -(2 * M_PI / mmax);
30              wtemp = sin(0.5 * theta);
31              wpr = -2.0 * wtemp * wtemp;
32              wpi = sin(theta);
33              wr = 1.0;
34              wi = 0.0;
35  #pragma omp for
36              for (m = 1; m < mmax; m += 2) {
37                  for (i = m; i <= n; i += istep) {
38                      j = i + mmax;
39                      tempr = wr * data[j - 1] - wi * data[j];
40                      tempi = wr * data[j] + wi * data[j - 1];
41
42                      data[j - 1] = data[i - 1] - tempr;
43                      data[j] = data[i] - tempi;
44                      data[i - 1] += tempr;
45                      data[i] += tempi;
46                  }
47                  wtemp = wr;
48                  wr += wr * wpr - wi * wpi;
49                  wi += wi * wpr + wtemp * wpi;
50              }
51              mmax = istep;
52          }
53      }
54  }
```

Listing A.2: Fast Fourier transform.

The code for a Fast Fourier transform is shown in Listing A.2. For ease of implementation, the first part of bit reversing the indices (lines 8 to 22) is not parallelized with OpenMP, as this step may be done directly when reading the input data. In order to avoid additional overhead when forking new threads for each stage of the butterfly, the whole next section (lines 24 to 53) is within one parallel region. In line 36 the index is incremented by 2, because the complex values are stored with the real part in even indices and the corresponding imaginary part in the following odd index.

As one can see in Figure 6.6 different communication patterns for the array with the data were used and compared.

A.3. Partial differential equation

```
1  double calculate(int N, double*** Matrix, int term_iteration) {
2      int i, j;            // local variables for loops
3      int m1, m2;          // used as indices for old and new matrices
4      double star;         // four times center value minus neighbouring
           ↪ values
5      double residuum;     // residuum of current iteration
6      double maxresiduum; // maximum residuum value of a slave in
           ↪ iteration
7      m1 = 0;
8      m2 = 1;
9      while (term_iteration > 0) {
10         double** Matrix_Out = Matrix[m1];
11         double** Matrix_In = Matrix[m2];
12         maxresiduum = 0;
13 #pragma omp parallel for private(i,j,star,residuum)
       ↪ reduction(max:maxresiduum)
14         for (i = 1; i < N; i++) {  //  over all rows
15             for (j = 1; j < N; j++) {  //over all columns
16                 star = 0.25 * (Matrix_In[i - 1][j] + Matrix_In[i][j -
                       ↪ 1] + Matrix_In[i][j + 1] + Matrix_In[i + 1][j]);
                       ↪ // calculate stencil
17                 star += fpisin_i * sin(pih * (double) j);  // add
                       ↪ disturbance function
18                 if (term_iteration == 1) {
19                     residuum = Matrix_In[i][j] - star;
20                     residuum = (residuum < 0) ? -residuum : residuum;
21                     maxresiduum = (residuum < maxresiduum) ?
                           ↪ maxresiduum : residuum;
22                 }
23                 Matrix_Out[i][j] = star;
24             }
25         }
26         i = m1; // exchange m1 and m2
27         m1 = m2;
28         m2 = i;
29         term_iteration--;
30     }
31     return maxresiduum;
32 }
```

Listing A.3: Code for solving a partial differential equation with a stencil code.

Listing A.3 shows the usage of OpenMP when solving a partial differential equation with a stencil code (line 15). As two matrices (lines 10 and 11) were used, there is no data-dependency within one iteration. OpenMP defines all values shared by default, so

the matrices are not explicitly declared shared in line 13.

This case uses the distributed array communication pattern explained in Section 5.6.2.

A.4. K-means clustering

```
1  int DIM; //dimension of data
2  int K; //number of clusters
3  int N; // number of datapoints
4  double threshold = 0.001f; // when to stop refining the cluster
       ↪ centers
5  double **objects; // input data
6  int *membership; // output
7  double **clusters; // output
8
9  void calculate() {
10     int i = 0, j = 0, l = 0, index = 0, master_loop = 0;
11     int *newClusterSize; // [numClusters]: number of objects assigned
           ↪ in each new cluster
12     int delta = 0; // % of objects change their clusters
13     int max_delta_allowed = floor(N * threshold);
14     double **newClusters; // [numClusters][numCoords]
15     int nthreads; // number of threads
16     int **local_newClusterSize; // [nthreads][numClusters]
17     double ***local_newClusters; // [nthreads][numClusters][numCoords]
18     nthreads = omp_get_max_threads();
19     /* pick first numClusters elements of objects[] as initial
           ↪ cluster centers*/
20     /* initialize membership[] */
21     /* initialize newClusterSize and newClusters */
22     // each thread calculates new centers using a private space, then
           ↪ thread 0 does perform an array reduction on them
23 #pragma omp parallel shared(objects, clusters, membership,
       ↪ local_newClusters, local_newClusterSize, delta, master_loop)
       ↪ private(i, j,l, index) firstprivate(N, K, DIM,
       ↪ max_delta_allowed) default(none)
24     {
25         int loop = 0;
26         int tid = omp_get_thread_num();
27         do {
28             // barrier after loop condition is checked
29 #pragma omp barrier
30 #pragma omp single nowait
31             { // only one needs to reset delta
32                 delta = 0.0;
33             }
34             /* each thread resets its local data */
35 #pragma omp for schedule(static) reduction(+ : delta)
36             for (i = 0; i < N; i++) {
37                 // find the array index of nearest cluster center
```

```
38                  index = -1;
39                  double dist, min_dist = DBL_MAX;
40                  for (j = 0; j < K; j++) {
41                      dist = 0;
42                      for (l = 0; l < DIM; l++)
43                          dist += (clusters[j][l] - objects[i][l]) *
                               ↪ (clusters[j][l] - objects[i][l]);
44                      // no need square root
45                      if (dist < min_dist) { // find the min and its
                           ↪ array index
46                          min_dist = dist;
47                          index = j;
48                      }
49                  }
50                  if (membership[i] != index)
51                      delta += 1.0;
52                  membership[i] = index;
53                  // update new cluster centers : sum of all objects
                       ↪ located within (average will be performed later)
54                  local_newClusterSize[tid][index]++;
55                  for (j = 0; j < DIM; j++)
56                      local_newClusters[tid][index][j] += objects[i][j];
57              } // end of parallel for
58                // implicit barrier
59  #pragma omp master
60              {
61                  // let the main thread perform the array reduction
62                  for (i = 0; i < K; i++) {
63                      for (j = 0; j < nthreads; j++) {
64                          newClusterSize[i] +=
                               ↪ local_newClusterSize[j][i];
65                          local_newClusterSize[j][i] = 0.0;
66                          for (l = 0; l < DIM; l++) {
67                              newClusters[i][l] +=
                                   ↪ local_newClusters[j][i][l];
68                              local_newClusters[j][i][l] = 0.0;
69                          }
70                      }
71                  }
72                  // average the sum and replace old cluster centers
                       ↪ with newClusters
73                  for (i = 0; i < K; i++) {
74                      for (j = 0; j < K; j++) {
75                          if (newClusterSize[i] > 1)
76                              clusters[i][j] = newClusters[i][j] /
                                   ↪ newClusterSize[i];
77                          newClusters[i][j] = 0.0; // set back to 0
78                      }
79                      newClusterSize[i] = 0; // set back to 0
80                  }
81              } // end master
82                // implicit barrier
```

```
83          } while (delta > max_delta_allowed && loop++ < 500);
84      } // end of #pragma omp parallel
85  }
```

Listing A.4: Code excerpt for K-means algorithm.

The K-Means algorithm is shown in Listing A.4.
`N` denotes the number of objects to cluster, `DIM` denotes the number of features for each datapoint (or the dimension of the points) and `K` is the number of clusters. In line 51 the number of datapoints, that switched clusters, is updated. In lines 54-57 the local cluster sizes and position of the new cluster center are updated. As a basic version of parallelizing the K-means algorithm was choosen, the cluster centers for each thread are reduced by the master thread (lines 59-81).
The function `find_nearest_cluster` determines to which cluster a given datapoint should belong to. Although the `membership` array is shared, there are no data dependencies between the threads as each thread updates the values only for the points assigned to him.

This case uses the distributed array communication pattern explained in Section 5.6.2.

A.5. Smith-Waterman algorithm

```
1  int Bw, // block width
2          Bh, // block height
3          N_a, N_b; // length of sequences
4  char[N_a] seq_a; // sequence A
5  char[N_b] seq_b;// sequence B
6  int[N_a+1][N_b+1]H;//matrix for dynamic programming
7
8  int main(int argc, char** argv) {
9      /* initialize input data */
10     // create threads
11 #pragma omp parallel default(none) shared(H, seq_a, seq_b, Bh, Bw)
       ↪ firstprivate(N_a, N_b)     \
12         shared(max)
13     {
14         int newRows, newCols, numOfDiags;
15         int T = omp_get_num_threads();
16         int my_tid = omp_get_thread_num();
17         // get the number of working threads
18         Bh = (int) ceil((double) N_a / T); // compute size of blocks
19         Bw = (int) ceil((double) N_b / T);
20         // last block will have less work
21         Bh = Bh < 1 ? 1 : Bh;
22         Bw = Bw < 1 ? 1 : Bw;
23         // calculate number of blocks
24         newRows = (int) ceil(((double) N_a) / Bh);
25         newCols = (int) ceil(((double) N_b) / Bw);
26         numOfDiags = newRows + newCols - 1;
```

```
27          // currently the pipeline below would not work otherwise
28          assert(newRows == T);
29          assert(newCols == T);
30          int this_row = my_tid; // not moving
31          int this_col = 0;
32
33          // create all blocks
34          for (int i = 0; i < numOfDiags; ++i) {
35              if (my_tid <= i && this_col <= newCols) // if this thread
                    ↪ should be active
36                      {
37                  computeBlock(this_row, this_col);
38                  this_col++; // next iter: next col
39              }
40  #pragma omp barrier
41          }
42      } // end parallel
43      /* search H for the maximal score (may be done paralell) */
44      /* (sequential) backtracking from H_max */
45  }
46
47  void computeBlock(Point block) {
48      const int row = block.row * Bh, col = block.col * Bw;
49      int Rows = MIN(N_a, row + Bh), Cols = MIN(N_b, col + Bw);
50      for (int i = row + 1; i <= Rows; ++i) {
51          for (int j = col + 1; j <= Cols; ++j) {
52              tmp[0] = H[i - 1][j - 1] + similarity_score(seq_a[i - 1],
                    ↪ seq_b[j - 1]);
53              tmp[1] = H[i - 1][j] - delta;
54              tmp[2] = H[i][j - 1] - delta;
55              // compute action that produces maximum score
56              H[i][j] = find_max_score(tmp);
57          }
58      }
59  }
```

Listing A.5: Code excerpt for the Smith-Waterman dynamic programming algorithm.

Listing A.5 shows the usage of OpenMP to parallelize the Smith-Waterman algorithm. The size and number of the blocks (as illustrated in Figure 3.6) is calculated in lines 14 to 26. The for loop (lines 34-41) iterates over all diagonales within the matrix. This means it loops over all regions that can be computed concurrently (also see Figure 3.6). Each thread will have one row of blocks assigned. Therefore each thread will start "entering" the pipeline only if it is the right time to do so (line 35).
The `computeBlock` function (lines 47-59) loops through a block of the score matrix and compute its contend according to Equation (3.1). The initialization of the matrix boundaries (`H[0][j]` and `H[i][0]`) with zeros is not shown (line 9).

For the score matrix the memory aware variant of distributed arrays (Section 5.6.2) was used.

A.6. Traveling salesman

```
1  double** adj; // adjacency matrix of the graph
2  int N; // number of nodes of the graph
3  double best_score = DBL_MAX; // stores the final minimum weight of
       ↪ the shortest tour
4
5  int main() {
6      /* init the graph */
7      #pragma omp parallel shared(best_score,adj)
8      int level = 1;
9      double* visited_nodes = malloc(N * sizeof(double));
10     visited_nodes[0] = 0; // start with node 0;
11     #pragma omp for
12     for (int i = 1; i < N; i++) {
13         visited_nodes[level] = i;
14         recursive_TSP(visited_nodes, level + 1);
15     }
16     // best_score now holds the cost of the best tour
17 }
18
19 recursive_TSP(double* visited_nodes, int level) {
20     if (level==N) { // reached a leaf
21         #pragma omp critical
22         {
23             if (score(visited_nodes)<best_score)
24             {
25                 best_score = score(visited_nodes)
26             }
27         }
28     } else {
29         for (int i=1;i<N;i++)
30         {
31             if (not_visited(visited_nodes,i))
32             {
33                 // compute the minimum cost of all solutions in the
                       ↪ branch that should be entered to see if it is
                       ↪ necessary to visit the branch
34                 if (bound(level)<best_score)
35                 {
36                     visited_nodes[level]=i;
37                     recursive_TSP( visited_nodes,level+1);
38                 }
39             }
40         }
41     }
42     // reset the node visited in this level
43     visited_nodes[level]=0;
44 }
```

Listing A.6: Code excerpt for solving the traveling salesman problem with a Branch and Bound algorithm.

Listing A.6 illustrates solving the traveling salesman problem with a Branch and Bound technique in a parallel way. Ignoring the OpenMP pragmas will result in the sequential program.
The function `not_visited` returns true if the given node (`i`) has not already been visited in `visited_nodes`. The function `bound` determines the minimal possible cost to complete the tour to use it as a bound. One simple way to bound the cost of such a tour is to take the number of not visited nodes plus one (for the way back to the origin node) times the cost of the least expensive edge in the graph. `score` determines the total cost of the given tour. If one also want to have the best possible tour as output, it is necessary to also update the best tour discovered yet at every update of the best score. This has to be done within the same critical region, so that the best tour always match with the best score.

In this case the data was duplicated (Section 5.6.4).

A.7. Test of task pragma

```
1  int N;// rows of matrix
2  int M;// cols of matrix
3  int RUNS;// number of runs
4
5  int main(int argc, char **argv) {
6      /* read input sizes and init Mat */
7  #pragma omp parallel shared(Mat)
8      {
9  #pragma omp single
10         {
11             for (int r = 0; r < RUNS; ++r) {
12                 for (int i = 0; i < N; ++i) {
13                     // create one task for each row:
14 #pragma omp task shared(Mat) firstprivate(i)
15                     {
16                         for (int j = 0; j < M; ++j) {
17                             Mat[i][j] += i * 10 + j;
18                         }
19                     }
20                 }
21                 // sync at the end of each run (so that no race
                       ↪ conditions inside the array may occur)
22 #pragma omp taskwait
23             }
24         } // implicit barrier
25     }
26     /* check if matrix has expected values */
27 }
```

Listing A.7: Test program that uses the OpenMP task pragma.

Listing A.7 shows the program used to evaluate the translation of the task pragma directive. For each row of the matrix a new task is created (lines 14-19 of Listing A.7).

For the matrix the memory aware variant of distributed arrays (Section 5.6.2) was used.

B. Adding a new communication scheme

As outlined in Section 5.6.1, in order to implement a new MPI communication pattern for an array or single value variable, four steps are required:

1. Implement the new communication functions in the communication library (refer Section 5.6.1).

2. Introduce them to the pass so that it may insert calls to them. It is suggested to follow the current design:

   - The reference to the `llvm::Function` object of each function that may be inserted by the pass is part of the `external_functions` struct.
   - In the `load_external_module_definitions` functions there is a `MATCH_FUNCTION` macro. This macro matches the function objects in the external module (communication library) by name to the given position of the `external_functions` struct. e.g.: `MATCH_FUNCTION(e->functions->free_distributed_shared_array_from_master,"_Z41free_distributed_shared_array_from_masterPvm")`.

3. Define and implement a new derived class from `SharedArray` or `SharedSingleValue`, depending on the type of variable. As explained in Section 5.6.1, the virtual functions have to be implemented by the new subclass. Additionally the new class should also overwrite the `classof` method. It is important to note, that the `classof` method of all superclasses might need to be adopted as well. This is needed, so that the LLVM `dyn_cast` operation can be used. Refer the source code comments for further explanation.
   Furthermore the new class need two constructors. Their implementation is quite straightforward analogous to the currently implemented constructors. For further information refer the comments in the source code.

4. Adapt the factory method so that it will create instances of this new class - also refer to Section 5.2. In the `parse_annotation` function the annotation is parsed and the communication type is then given to the `CreateArray` or `CreateSingleValue` function respectively, where it is used in the switch instruction to instantiate the new `SharedVariable` object from the correct class.

C. Additional Listings

```
1  void mpi_load_shared_struct(void *buffer, struct mpi_struct_ptr
       ↪ *struct_ptr, MPI_Win *win)
2  {
3      if (struct_ptr->rank != -1)
4      {
5          MPI_Win_lock(MPI_LOCK_SHARED, struct_ptr->rank, 0, *win);
6          MPI_Get(buffer, (int)struct_ptr->size, MPI_BYTE,
               ↪ struct_ptr->rank, struct_ptr->displacement,
               ↪ (int)struct_ptr->size, MPI_BYTE, *win);
7          MPI_Win_unlock(struct_ptr->rank, *win);
8      }
9      else
10     {
11         printf("ERROR: loading from nullptr\n");
12     }
13 }
```

Listing C.1: `mpi_load_shared_struct` function.

List of Figures

List of Listings

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang 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.

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Ort, Datum	Unterschrift