A Golang Wrapper for MPI

— Project: parallel programming —

Scientific Computing
Department of Informatics
MIN Faculty
University of Hamburg

Submitted by: Alexander Beifuss
E-Mail-Address: 7beifuss@informatik.uni-hamburg.de
Matriculation number: 5953026
Study course: Informatics (M.Sc)

Submitted by: Johann Weging
E-Mail-Address: 8weging@informatik.uni-hamburg.de
Matriculation number: 6053290
Study course: Informatics (M.SC)

Advisor: Dr. Julian Kunkel

Hamburg, March 4, 2014

Abstract

This project aims to implement bindings of the Message Passing Interface for Googles programming language Go. Go is a young, clean, to native machine code compiling programming language which has the potential to gain ground inside the scientific community and the field of high performance computing. It offers a variety of concurrency features that can be used for parallel programming on shared memory architectures. There all ready exists bindings for different data formates like HDF5 or approaches for GPGPU-Computing with OpenCL or CUDA. This project will enable Go to be run on compute clusters and parallel programming over the network in general. The project uses cgo to wrap Go around existing MPI implementations like OpenMPI and MPICH2. The final implementation of the bindings is than benchmarked against C to determine it's usefulness and potential. Go, like expected is slower than C but there are still a lot of possibilities for improvements to catch up with the performance of C. The current bindings support a C like interface to MPI according to the standard with only minor changes. Supported is MPI version two and the implementations OpenMPI and MPICH2. In the future support for MPI version 3 and other implementations will follow.

Contents

1. Motivation

The motivation to develop Go bindings for MPI is to push another programming language to compute clusters and scientific computing. Go is a easy to learn language, suitable for *HPC* users and scientist that are not computer scientist. The default language C used for scientific computations has a lot of pitfalls and can be quiet tedious at some point.

It is a fast language since it compiles to native machine code and dose not require a virtual machine like Java or is interpreted like Python. In addition Go has a already builtin concurrency features which can be utilized for parallel computation. This makes Go a interesting tool for high performance computing and parallel computation in general.

Go is easy to wrap around existing C libraries. Where writing a MPI implementation in pure Go would take a lot of effort and will unlikely match the performance of an optimised and well tested existing MPI implementation. Furthermore multiple implementations can be wrapped and Go can take benefit of proprietary implementations and hardware.

Our report is structured as follows. In the next section we introduce the MPI (Message Passing Interface) standard and Golang (Google language) to provide the reader with some basics. The introduction is followed by a short description of the initial situation which prevailed when we started our project. In section 4 we briefly state our project goals. Section 5 deals with the implementation of our wrapper functions. Afterwards we describe the benchmark tool and present its results in section 6. Finally we give a conclusion as well as an outlook in section 7.

2. Introduction

Before directly starting with the implementation details of our wrapper functions, we want to give the reader the chance to become familiar with the MPI standard and Golang system programming language. It will show which technologies where used to successfully implement the bindings and show how to use them. This section splits up into two subsections one for each. The first subsection gives a rough overview of MPI and exemplifies what it is good for. The second subsections deals with Googles new programming language. At the beginning we provide the historical background of Golang. Then we switch to a part that is written in a tutorial like fashion that explains the peculiarities of Golang.

2.1. Introduction into MPI (3)

MPI is used on compute clusters for parallel programming over the network. The examples used in this section are written in C and not in Go. The Go examples will follow later in this document (see 2.3).

2.2. Basic Concept

The basic concept is like the name states it, message passing. Meaning different processes are communicating with each other by passing data as messages around. Each process is identified by a unique identifier called rank.

Listing 2.1: Simple MPI example

```
1  char message[100];
2  int rank = MPI_Comm_rank(MPI_COMM_WORLD);
3
4  if (rank == 0) {
5      message = "Hello Wolrd!\n"
6      MPI_Send(message, 100, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
7  } else {
8      MPI_Status status;
9      MPI_Recv(message, 100, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &status);
10     printf(message);
11 }
```

Listing 2.1 shows a very simply MPI program. Rank 0 will send the message *Hello World!* to rank 1 and rank 1 will print the message. The parameter used are the data itself, the count of elements to send, the datatype, where to send the data to / receive

form, a tag to identify the message and a communicator containing information about which processes can speak with each other. The receive statement additionally has a status parameter which contains information about the received message.

Listing 2.2: Simple MPI example for non-blocking communication

```
1  char message[100];
2  MPI_Request request;
3  int rank = MPI_Comm_rank(MPI_COMM_WORLD);
4
5  if (rank == 0) {
6      message = "Hello Wolrd!\n"
7      MPI_ISend(message, 100, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &request);
8      MPI_Wait(&request, &status)
9  } else {
10     MPI_Status status;
11     MPI_IRecv(message, 100, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &request);
12     MPI_Wait(&request, &status)
13     printf(message);
14 }
```

Listing 2.2 shows the usage of non-blocking commutation. Non-Blocking communication is used to over leap computation and communication to increase the parallelism of the program. Another aspect is to avoid dead locks. The program in this example dose the same as the program in listing 2.1. The send and receive function are replaced with the intermediate equivalent *MPI_ISend* and *MPI_IRecv*. These function require a additional parameter, a request object, which is used to determine if the call has finished. To determine if a non-blocking function call has finished one can use *MPI_Wait*. This call blocks until the corresponding intermediate function hast finished and returns.

Listing 2.3: Simple MPI example for collective communications.

```
1  char message[100]
2  int rank = MPI_Comm_rank(MPI_COMM_WORLD)
3  if (rank == 0) {
4      message = "Hello from rank 0!\n";
5  }
6
7  int MPI_Bcast(message, 100, MPI_CHAR, 0, MPI_COMM_WORLD);
8  if (rank != 0) {
9      printf(message)
10 }
```

Listing 2.3 shows the simple usage of collective communication. The code snippet takes the message from rank 0 and broadcast it to every one else. Every one else than prints the message. The rank which sends the message is determined by root rank passed as 4th argument of the broadcast function. In this case the rank 0. Collective communications must be called by every process inside the communicator passed to the function. In this example *MPI_COMM_WORLD* is passed to the function, this means every process that runs this program needs to call this function. The call blocks until every process has received the message from rank 0. This will lead to poor performance if there is a load

imbalance in the program since all processes have to wait for the slowest to complete the collective call.

2.2.1. Non-Blocking Collective Communication

Non-Blocking collective communication is a new set off function introduced in MPI 3 [MPI12]. They are very useful since collective operations are expensive. Like the point to point non-blocking communication the function name prepends a *I* to the operation name to indicate that it's a intermediate function. The intermediate equivalent of *MPI_Bcast* is *MPI_IBcast*. The function arguments are mostly the same as the blocking function call, all the the intermediate function needs is a additional request object to determine if the call has finished. These request can than be passed to *MPI_Wait* or *MPI_Test* to check if the call has finished.
One special case is the non-blocking barrier *MPI_IBarrier*. Like all non-blocking function *MPI_IBarrier* the function returns instantly. This is useful if the synchronization is needed but the program can continue to do other computation. The synchronization happens asynchronous. At some point the program needs to ensure that the synchronization has finished using a function like *MPI_Wait*. In the best case synchronization has already happened in the background when calling *MPI_Wait* and the program can continue to run.

2.2.2. Neighbourhood Collective Functions

Another addition in MPI 3 is the introduction of neighbourhood collectives. These functions operate on a topology that is created first. A topology is a way to model which compute nodes are neighbours of each other and thus can directly communicate with each other.

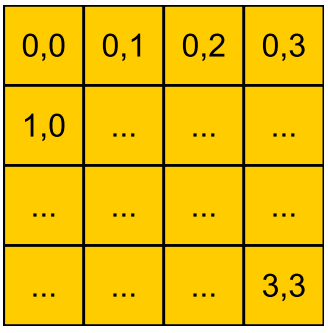


Figure 2.1.: A simple topology to be used with MPI.

Listing 2.4: The topology of figure 2.1 implemented in MPI.

```
1
2 MPI_Comm topologyComm;
3 MPI_Cart_create(MPI_COMM_WORLD, 2, {4,4}, {0,0}, 1, &topologyComm);
4 int rank = MPI_Comm_rank(topologyComm);
5
6 int recvBuf[4] = {-1,-1,-1,-1};
7 MPI_Neighbor_allgather(rank, 1, MPI_INT, recvBuf, int recvcount,
    ↪ MPI_INT, topologyComm);
```

Lets consider the topology of figure 2.1. It is a two-dimensional array with four by four processes. Listing 2.4 shows the same topology modelled with MPI. To model a n-dimensional array one can use the function *MPI_Cart_create*. The first parameter is the old communicator, which indicates which processes should be involved in the new topology. The second parameter is the count of dimensions of the array. The third parameter is a array providing the size of the dimensions. The fourth parameter is a array indicating if a dimension should be periodic. The fifth parameter tells the function if the ranks are allowed to be reordered. The last parameter is the topology presented by a communicator. The program creates the new topology on top of the *MPI_COMM_WORLD* communicator. And than calls *MPI_Neighbor_allgather* to obtain the the ranks of its neighbours. Since one rank has a maximum of four neighbours the receive buffer can store four elements. The data of the first neighbour is stored at the first position, the second at the second and so on. With the function *MPI_Dist_graph_create* MPI is able to create arbitrary topology, modelling all kinds of graphs. The neighbour functions functions of MPI don't lead to a significant performance improvements but they are used to make the program code cleaner and easier to read.

2.3. Introduction into Golang

In this section we provide some basic information about Google's new programming language called Go (also known as Golang). Rather than going to much into detail and covering every aspect of Golang, we give a short and compact introduction to bring the readers to a common level. For those readers who want to delve much more into detail, we suggest the official Golang documentation [Go] or one of the few books ([Bal12] [Chi12] [FB10], [MÏ1], [Sum12]) that are available. On the other hand, readers who are already familiar with Golang might feel free skip this section — and fall back as needed.

2.3.1. History

In September 2007 Ken Thompson, Rob Pike and Robert Griesemer began with the design of Google Go at Google Inc as a free time project. In the mid of 2008 the project had become a full-time project and Ross Cox joined the team. On November 10th 2009 Golang has been officially announced and became a public open source project. After several releases in 2011, the first stable long term major release *go1* has been released on March 28th 2012. Three minor revisions followed in the same year. The current version *go1.1* has been released on May 13th 2013 the minor revision *go1.1.1* followed one moth later on June 13th 2013.

2.3.2. Variables, Function Calls and Return Values

Since Golang's Syntax differs form those most of the readers might be familiar with from other programming languages (like C, Java or Python), we will present some code examples and depict some of the major differences. Like nearly every guide, we start with a 'hello world' example (listing 2.5).

Listing 2.5: 'Hello world' example in Golang

```
1 package main
2
3 import "fmt"
4
5 func main() {
6   fmt.Println("Hello, world!")
7 }
```

In the first line of the example there is a *package* statement. It is always the first statement in a Golang source file and it is used by Golang to declare the default package(name) for later imports in external projects. Since the example should be build as an executable (and not as a package), the name of the package should conform to 'main'. The third line shows an *import* statement. The keyword *'import'* is followed by a string that specifies which packet should be imported. In this case the imported the package is 'fmt' which exports the I/O functionality needed to print out text on the shell. The 'main' function begins in line 5. Unlike in the programing language C, Golang needs a special keyword (namely *'func'*) to indicate the beginning of a function declaration. To the right of this

keyword, one has to provide the signature of the function which should be declared. Here the 'main' function is declared and in this case it neither takes arguments nor returns something — the 'main' function is like in most other programming languages the entry point of the program. The curly brackets after the function signature indicates the beginning of the function definition. Since the function body starts with a curly bracket, it also has to be closed with a corresponding curly bracket (line 7). The lines which are surrounded by such a pair of bracket, build the function defining code block. In the 'hello world' example the function *'Println'* is called with the string 'Hello , world !' given as an argument. The prefix 'fmt.' tells Golang, that 'Println' is part of the 'fmt' package (which already has been imported in line 3), so Golang is able to find the function that should be called. Some readers might wonder why 'Println' starts with a capital letter, since in other languages it is convention to begin a function name with lower case. The reason for this is, that in Golang the beginning capital letter is used indicate a function that is callable from outside the package (the function is public) — on contrary, a function name beginning with a lower case letter indicates a private function, that is just callable from inside the package. Additionally, it is to mention that the Golang compiler in fact expects a semicolon at the end of each expression. Looking at the listing one will not find a single semicolon. Strictly speaking programmers are even discouraged to write those semicolons by themselves. An integrated set of rules keeps track of proper code formating and automatically inserts such semicolons right before compiling. One can prove this by adding a semicolon at the end of line 6 and apply Golang's formatter tool *gofmt* to the code — the semicolon is going to be removed in the output, but the output is still compilable as well as the code with the additional semicolons. To run the example, one could use the *Go Tools* and either build end execute the program in a single step with the command 'go run <sourcefile.go>' or first build the executable with 'go build <sourcefile.go>' and then execute it with './<sourcefile>'.

The listing 2.6 depicts a modified version of the former 'hello world' example. It is extended by an additional function named 'assembleSalutation' which expects two arguments of the type string. This function concatenate each of these arguments with the static part of our salutation and then returns both (yes, both!) assembled strings to the main function.

First of all we will examine line 5 which contains the signature of our new function. Between the brackets we find two arguments namely 'nameOne' and 'nameTwo' which are of type string. Contrary to most other popular programming languages, the programmer has to provide the type directly behind the argument's name. Since both arguments are of the same type, Golang allows to reduce the typing overhead by declaring the type just once after the last argument — this also holds for more than two arguments of the same type. Like Python do Golang provides the ability to return multiple values. In the function signature the tuple of return values is defined right after the argument list — for single return values one can omit the brackets. But as our function returns two values return has to name exact two values which must be separated by a colon (see line 6). Dealing with two or more return values of the same type requires some care since the

order of providing the values is important. The ability to return more than one value does not increase the expressiveness of a programming language but it helps to avoid packing data into structs or dealing with output parameters and therefore supports the programmer in writing cleaner and more readable code.

Listing 2.6: Extended 'Hello world' example in Golang

```
1  package main
2
3  import "fmt"
4
5  func assembleGeetring(nameOne, nameTwo string) (string, string) {
6    return "Hello, " + nameOne + "!", "Hello, " + nameTwo + "!"
7  }
8
9  func main() {
10   var jane string
11   jane = "Jane"
12   var jack = "Jack"
13
14   saluationOne, saluationTwo := assembleGeetring(jane, jack)
15   fmt.Println(saluationOne)
16   fmt.Println(saluationTwo)
17 }
```

Now let us have a look at the main function. In line 10 a string variable named "Jane" is declared by using the keyword *var* followed by the variable name followed by its data type. In line 11 the sting value "Jane" is assigned to this variable. Line 12 depicts how to declare a variable and directly assign a value to it in a single step. The attentive reader noticed that we do not define the data type in this variable declaration — doing the declaration and assignment in a single step allows to omit the type as the Golang compiler can deduce the type from the value which should be assigned to the variable. An alternative version of performing the declaration and the assignment in a single step is shown in line 15. Instead of using the assign operator *'='* we make use of another operator *':='* which declares a variable and directly assigns a value to it. As one can see this operator also allows multiple declarations and assignments, since it is allowed to have multiple variables on the left side of the operator which is necessary because a function (e.g. 'assebleGreeting') might have multiple return values.

2.3.3. Data Types

In this subsection we cover the data types of Golang. After giving a short introduction to the numeric types, we have a closer look to the built-in collection types 'array' and 'slice' as well as the strings type since these are the interesting high-level types.

Regarding the data types, it is first to mention that each type in Golang is distinct. In Listing 2.7 a simple program is listed that performs compare operations with different integer types.

Listing 2.7: Example of distinct types in Golang

```
1  package main
2
3  import (
4    "fmt"
5  )
6
7  func main() {
8
9    a := int(42)
10   b := int32(42)
11   c := int64(42)
12
13   if a == b {
14     fmt.Println("a = b")
15   }
16
17   if a == c {
18     fmt.Println("a = c")
19   }
20 }
```

Trying to compile this program will result in compiler errors, because *int, int32* and *int64* are pairwise not comparable. This means the type conversion has to be performed explicitly. Listing 2.8 shows the two modifications that are necessary to make the program compilable.

Listing 2.8: Example of type conversion

```
1    if (int32)(a) == b {
2      fmt.Println("a = b")
3    }
4
5    if (int64)(a) == c {
6      fmt.Println("a = c")
7    }
```

While the actual program would do its job for values fitting in a 32 bit signed representation higher values might cause an overflow when conversing *int* to *int32*. Using the *unsafe*-package and the *Sizeof*-function as shown in listing 2.9, we can test the sizes of *int32* and *int64*. As expected *int32* has a size of 4 Byte while *int64* is with 8 Byte two times as big as *int32*. The exact size of *int* depends on the system and is either 4 or 8 Byte (this also holds for the unsigned integer types).

Listing 2.9: Example of type conversion

```
1  package main
2
3  import (
4    "fmt"
5    "unsafe"
6  )
7
8  func main() {
9
10   fmt.Println(unsafe.Sizeof(int32(42)))
11   fmt.Println(unsafe.Sizeof(int(42)))
12   fmt.Println(unsafe.Sizeof(int64(42)))
13 }
```

In addition to the signed and unsigned integer types, Golang provides also built-in numeric types for floating-point and complex numbers (native support for complex numbers might be very favorable for scientific computing). A complete list of all numeric types is given in table 2.1.

uint8 / byte	8 bit unsigned integer
uint16	16 bit unsigned integer
uint32	32 bit unsigned integer
uint64	64 bit unsigned integer
int8	8 bit signed integer
int16	16 bit signed integer
int32 / rune	signed integer
int64	64 bit signed integer
float32	32 bit floating-point numbers
float64	64 bit floating-point numbers
complex64	complex numbers with float32 real and imaginary parts
complex128	complex numbers with float64 real and imaginary parts
uint	either 32 or 64 bit unsigned integer
int	either 32 or 64 bit signed integer
uintptr	unsigned integer large enough to store a pointer

Table 2.1.: Numeric types in Golang

Unlike C Golang provides a built-in type for strings. The string type is realized as a sequence of bytes (*uint8*) interpreted as series of UTF8 characters. The length of a string can be obtained using the *len*-function. Accessing a specific element of a string *s* is possible using the indexes between 0 and *len(s)-1*. In Golang strings are very flexible and it is also possible to directly access substrings by using the slice operator (we cover this later). In oder to manipulate strings Golang also provides a very powerful package with the name 'strings'. Listing 2.10 gives a short overview of how to handle the string type.

Listing 2.10: Using strings in Golang

```
1  package main
2
3  import "fmt"
4  import "strings"
5
6  func main() {
7
8    var str string = "Hello, world"
9    // str[0] = 'b' invalid, strings are immutable
10   for i := 0; i < len(str); i++ {
11     fmt.Printf("%c", str[i])
12   }
13   fmt.Printf("\n")
14   fmt.Println(str[3 : len(str)-3])
15   fmt.Printf("%q\n", strings.Split(str, ","))
16 }
```

Golang also provides array types. An array is a sequence of elements of a particular type (this type is named *element-type*) which are located in *continuous* block of memory. In Golang the array type is defined by the element-type and the number of elements (called the *length* of an array). Since the length is part of the type, it must evaluate to a constant integer during compile time. Like with strings, the *len*-function can be used to peek an array's *length*.

Listing 2.11 introduces the usage of arrays in Golang.

Listing 2.11: Using arrays in Golang

```
1  package main
2
3  import "fmt"
4  import "unsafe"
5
6  func callByValue(a [9]int, name byte) {
7    for i, elem := range a {
8      fmt.Printf("%c[%d]: %d\n", name, i, elem)
9    }
10 }
11
12 func callByReference(b *[9]int, name byte) {
13   for i := 0; i < len(b); i++ {
14     fmt.Printf("%c[%d]: %d\n", name, i, b[i])
15   }
16 }
17
18 func main() {
19   a := [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8}
20   callByReference(&a, 'a')
21
22   b := new([9]int)
23   b[1] = 1
```

```
24    b[2] = 2
25    callByValue(*b, 'b')
26
27    var c [3][3]int
28    c[0][0] = 0
29    c[1][0] = 3
30    c[2][0] = 6
31    callByReference((*[9]int)(unsafe.Pointer(&c)), 'c')
32 }
```

Line 19, 22 and 27 illustrate three different variants of declaring array variables in Golang. In line 19 the variable *a* is not just declared but also initialized. Since the initializer list holds a constant number of elements the compiler can derive the length of the array, so the programmer does not need to provide the length explicitly. To indicate that the compiler should derive the length it is sufficient to write three dots instead of the length while providing the type. It might be a bit confusing but unlike in C, arrays in Golang are values. This means that if an array is passed to a function the whole array will be copied (call by value). Sometimes it is wise to pass a reference (call by reference) in order to avoid too much overhead. To obtain the address one can use the address operator (*&*) as line 20 indicates.

In line 22 the built-in function *new* is used to allocate memory for an array of three *int*. The special about *new* is that is allocates a block of zeroed storage and returns the reference. Therefore, *b* is a pointer to an *int* array of the length three where each element has initially the value zero. The next two lines show how values are assigned to the second and the third element of the array. To dereference a pointer one can use the dereference operator *. Referencing and dereferencing in Golang behaves similar to referencing and dereferencing in C but there are no pointer arithmetics in Golang.

In line 27 the variable *c* is declared to be a two dimensional array of *int* (more dimensions are also possible). Like before the array is initialized with zeros. The whole array is represented by a continuous block of memory and therefore it is possible to cast a reference of the two dimensional array into any shape (e.g. a reference of an one dimensional array of the length nine, see line 31). Although arrays are very useful for allocating continuous blocks of memory they are not very often used in high-level programs. This is because they are very impractical regarding the fact that it is tedious to write functions for specific array types (sizes). Instead arrays are used as building blocks for so called slices. Slices are data structures that consist of three different items.

1. data pointer into an array
2. length
3. capacity

While arrays can be allocated with *new* slices (and some other high-level types) has to be allocated with *make*. The difference between both functions is that *new* just allocates memory and *make* additionally performs some initialization. The initialization is necessary because the items of the data structure slices needs to be set before the slice can be used (e.g. allocating the underlying array(s)). A slice can be considered as a wrapper for arrays that holds the length in its structure (and not in its type definition). This allows it to write functions that accepts data(-blocks) with arbitrary length.

Listing 2.12 depicts how slices and arrays are connected and that shaping can be realized by simple casting.

Listing 2.12: Handling slices in Golang

```
1  package main
2
3  import "fmt"
4  import "unsafe"
5
6  func main() {
7    a1 := make([]int, 3, 10)
8    fmt.Println(a1)
9    a2 := a1[0:cap(a1)]
10   a2[1] = 1
11   fmt.Println(a2)
12   fmt.Println(a1)
13
14   b1 := make([][3]int, 2, 3)
15   b1[1][0] = 42
16   fmt.Println(b1)
17
18   b2 := *(*[][2]int)(unsafe.Pointer(&b1))
19   fmt.Println(b2)
20
21   b2 = b2[0:cap(b2)]
22
23   b2[1][1] = 23
24   fmt.Println(b1)
25   fmt.Println(b2)
26 }
```

Line 7 shows how built-in allocation primitive *make* is used to build a slice. The first parameter defines the type of the slice. In this case a slice of *int* should be allocated. The second parameter specifies the length and the third parameter the capacity of the slice. Internally the following happens: A continuous block of memory for 10 *int*s is allocated. Then the address of the array is assigned to the data pointer of the slice, the length is set to 3 and the capacity field gets the value 10. Since the slice type comprises the element type the address for element can be calculated by offsets. So each element is accessible via an index. Printing the complete slice will result in the output of an three-item array where each data element is zero. Now it should be easy to understand where the name *slice* comes from. A *slice* is just a data segment. In this case the first

three elements of an underlying array that has in total 10 elements. The slice operator (see line 9) is used to create new slices from existing slices or arrays (*cap* is a built-in function to obtain the capacity of a slice). Although *a1* and *a2* are different arrays they share the same memory since both data pointers point to the same array. This means changing an element of *a1* also effects the representation of *a2* iff the changed element is part of intersection of *a1* and *a2*.

In line 14 a new slice (*b1*) is created but this time the element type is *[3]int* and the length of the slice is 2, so the slice has a total number of 6 elements. Printing this slice will unveil its two-dimensional structure of two arrays each with a length of 3. In line 18 the pointer to the original slice is converted into a pointer of a slice of integer arrays of the length 2. Afterwards, this pointer is dereferenced and stored in the variable *b2*. The intention of this line is to reshape the slice. Printing *b2* shows that this new slice now addresses 2 *int* arrays each of the length 2. The length of the slice has not been changed because casting just effects the type and since the length is not part of the type it remains. To correct the slice length one could use the slice operator as shown in line 21. To proof that the new slice *b2* shares the data with *b1*, the 4th entry is altered in line 23. Then, both slices are printed out to verify that the value of the 4th element of each slice is now 23 instead of 42.

While the casting technique presented in listing 2.12 is nice to understand the structure of slices (and eventually suitable for libraries), it is definitely bad style in real applications and should be avoided. Listing 2.13 demonstrates a much better way for productive applications to build dynamically a two-dimensional slice with a continuous block of memory.

Listing 2.13: Dynamic allocation of a two-dimensional array

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    x, y := 4, 2
7
8    flat := make([]int, x*y)
9    dim2 := make([][]int, x)
10
11   for i := range dim2 {
12     dim2[i] = flat[i*y : (i+1)*y]
13   }
14 }
```

Let x and y hold the lengths for dimensions. First a flat *int* slice of the length $x*y$ is allocated. Remember: the allocated space is a continuous block in memory. Afterwards, a second slice is created which is a slice of x slices. Looping over the elements of *dim2* allows to assign successively slices of the flat slice (with length y).
If things getting more complex (like in real world applications) it is convenient to structure data that belongs together. These days most programming languages follow

the paradigm of object-orientation and encapsulate data and corresponding functions in so called classes. Golang is different. Strictly speaking it is a procedural language like C and does not follow the concept of classes. However, Golang provides structures (called *structs*) to combine data. Listing 2.14 shows how to define and initialize *structs*.

Listing 2.14: Structuring data in Golang

```
1  package main
2
3  import (
4    "fmt"
5  )
6
7  type tube struct {
8    radius float64
9    height float64
10 }
11
12 type block struct {
13   length float64
14   width  float64
15   height float64
16 }
17
18 func main() {
19
20   x := new(struct{ f1, f2 int })
21   x.f2 = 2
22
23   t := tube{1.0, 2.0}
24   b := new(block)
25   b.width = 1.0
26
27   fmt.Println(x)
28   fmt.Println(t)
29   fmt.Println(b)
30 }
```

In line 20 the allocation primitive *new* is used to allocate memory for an anonymous *struct* that combines two fields of the type *int*. The pointer to the new structure is assigned to the variable *x*. Using the dot-notation (line 21) one can access the fields within the structure. The dot-notations works for values as well as for pointers since Golang is able to evaluate the type of a variable and if necessary able to perform the dereferencing. Besides creating anonymous *structs* as in line 20 it is possible to define a type based on *structs*. In the lines 7 to 10 and 12 to 16 it is shown how the keyword *type* is used to define data types. The name for the new type occurs right behind the keyword *type*. The name is followed by the type that should be associated with this name — in this case an anonymous *struct*. Here a new type *tube* is defined which is a *struct* of two named fields radius and height both of the type float64. In line 23 the variable *t* is declared to be of the type tube and then an initializer-list is used to initialize the fields

within the structure. Heads up: The order of providing the values in the initializer-list is important and has to correspond to the order within the type definition.

In line 24 the allocation of a new variable is done using *new* and a named type. Here it is not possible to provide an initializer-list. All fields are initialized with zeros and then the fields have to be set manually using the dot-notation as shown in line 25.

2.3.4. Type specific functions

While typical object-oriented programming languages encapsulates the functionality together with the data in classes Golang does no follow tis concept. Golang works different. Generally it is procedural but it allows to write functions associated with a specific type and therefore supports an object-oriented style of programming. Listing 2.15 is an extension of listing (2.14) and depicts the association of functions and types.

Listing 2.15: An example of writing type specific functions in Golang

```
1  package main
2
3  import "fmt"
4  import "math"
5
6  type Tube struct {
7    radius float64
8    height float64
9  }
10
11 type Block struct {
12   length float64
13   width  float64
14   height float64
15 }
16
17 func (t Tube) Volume() float64 {
18   return 2.0 * math.Pi * t.radius * t.height
19 }
20
21 func (b Block) Volume() float64 {
22   return b.length * b.width * b.height
23 }
24
25 func main() {
26   t, b := Tube{1.0, 1.0}, Block{1.0, 1.0, 1.0}
27   fmt.Println(t.Volume(), b.Volume())
28 }
```

In line 17 and following as well as in line 21 and following a new notation of writing type specific functions is introduced. The only difference to general functions is that the type to which this function should be bound is provided between the keyword *func* and the function's signature. It is possible to bind the calling object to a local variable.

So the object and its fields are accessible within the function definition. Also, one can omit this variable but then the type bound function can not operate on the specific data of an object. That would not be very meaningful and come close to what some pure object-oriented programming languages introduce as *static* functions where classes are not only templates for objects but also part of the namespace. In order to call such an associated function the dot-notation is used again which is typical for object-oriented programming languages. From now on we consider the term *'object'* as the value of a variable with a particular type and the set of associated functions.

2.3.5. Interfaces

The concept of interfaces is where Golang really differs form common object-oriented programming languages. While inheritance is typically a key-feature of class based object-orientation that enables polymorphism Golang neither has a real class concept nor allows to derive attributes and / or functionality from existing types and their associated functions. However, Golang allows to write functions that can handle different types even though there is no superclass. This is possible because Golang provides a concept that is called interface. An interface is a set of functions and each object who's type provides all functions required by the interface automatically implement the interface. This means the type of an object can be deduced by its implemented functions. Such a technique is often called *'duck-typing'*. This name comes from a famous poetry: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck." (James Whitcomb Riley)

In listing 2.16 the old example from listing 2.15 is extended and presents the usage of *interfaces* in Golang.

Listing 2.16: A simple example of using interfaces

```
1  package main
2
3  import "fmt"
4  import "math"
5
6  type Tube struct {
7    radius float64
8    height float64
9  }
10
11 type Block struct {
12   length float64
13   width  float64
14   height float64
15 }
16
17 type Modeler interface {
18   Volume() float64
19 }
```

```
20
21 type Model struct {
22   mod Modeler
23 }
24
25 func (t Tube) Volume() float64 {
26   return 2.0 * math.Pi * t.radius * t.height
27 }
28
29 func (b Block) Volume() float64 {
30   return b.length * b.width * b.height
31 }
32
33 func volumeDifference(v1, v2 Modeler) float64 {
34   return v1.Volume() - v2.Volume()
35 }
36
37 func main() {
38   t := Tube{1.0, 1.0}
39   b := Block{1.0, 1.0, 1.0}
40   m := Model{t}
41   fmt.Println(m)
42   fmt.Println(volumeDifference(m.mod, b))
43 }
```

Mainly this example is extended by three definitions. The first one is the definition of an *interface* with the name *Modeler* (beginning at line 17). The definition says that everything that implements the function *Volume()* is a *Modeler* (interfaces can also be considered as types). The second definition, begins at line 21 and is the definition of the type *Model* that has a field of the type *Modeler*. The third definition defines a global function named *volumeDifference* that requires two objects implementing *Modeler* and then calculates the difference between the *float64* values of both objects which are returned when *Volume()* is called.

Besides these definitions, there are some changes in the main function. In line 40 the variable *m* is declared and initialized with the value of *t*. Since *t* has the type *Tube* which implements *volume* and therefore fulfills the *interface* named *Modeler* it is convenient to make this assignment. Printing out *m* shows that it is a *struct* which contains another *struct* with two fields both with the value 1. So it is possible to use *interfaces* as a kind of polymorph type within other type definitions. Furthermore, *m.mod* is known to be something that fulfills *Modeler* and *b* is from type *Block* which implements *Volume()* and therefore fulfills *Modeler* as well. This mean both values can be used as parameters for *volumeDifference(...)* (see line 42).

The benefit of resigning the concept of classes and instead using *'duck-typing'* lies in the reduction of keywords. Golang consequently tries to avoid as many keywords as possible (and they do it very well) to keep the language simple. So, the programmer is not bothered with typing work. But there is also a pitfall that comes along with

interfaces. The programmer has to be careful, because a bad chosen function name can accidentally make a type implementing an interface.

2.3.6. Concurrency Features

On of the major goals of Golang is to provide an easy way to handle concurrency (and even parallelism). Inspired by Tony C. Hoare's work "Communicating Sequential Processes" [Hoa78] Golang picked up the idea of combining *'parallel composition of sequential processes'* with *'guarded commands'*. The result is an integrated high-level concurrency concept based on two parts: Coroutines and synchronization by communication.

Goroutines

The implementation of coroutines in Golang is called *goroutine*. *Goroutines* can be considered as a lightweight version of threads. Compared to real OS threads they are really lightweight because they can coexist in a single OS thread. *Goroutines* are completely managed by the *Go runtime*, which is part of every go program. This saves a lot of context switches at the OS-level and reduces the overhead of creating threads. Therefore *goroutines* are even convenient for very small routines. Listing 2.17 shows how *goroutines* are used.

Listing 2.17: A simple example of using goroutines

```
1  package main
2
3  import "fmt"
4  import "time"
5
6  func sequential() {
7    for i := 0; i < 10; i++ {
8      fmt.Println(i)
9    }
10 }
11
12 func concurrent() {
13   for i := 0; i < 10; i++ {
14     go func(number int) {
15       fmt.Println(number)
16     }(i)
17   }
18 }
19
20 func main() {
21   go sequential()
22   concurrent()
23   time.Sleep(1)
24 }
```

In this listing there are two additional functions next to the main function. The first one is the function definition for *sequential*. This definition starts in line 6 and its semantics

is just to loop 10 times over a print statement that prints the actual value of the counter variable. In line 21 the program calls this function but it is called in a special way! In front of the function name there is the small keyword namely *go*. This keyword tells the Go-runtime to create a new *goroutine* and run the called function within it. Since the go-runtime also runs in a *goroutine* and the go-runtime is responsible for spawning the main function there are now two active *goroutines*. Each of this *goroutines* can be pictured as a separate and independent path of execution. After creating the new *goroutine* the go-runtime has to schedule the *goroutines*. The number of maximal parallel working *goroutines* depends on the system. On a multi core machine this means it is possible that both *goroutines* run in parallel. In case of a single core machine the scheduler has to pick an active *goroutine*, where active means the *goroutine* is not blocked (we cover this later when we talk about *channels*). Now things are getting fuzzy! Since the scheduler is an implementation detail, it is not possible to make assumptions about the order in which the go-runtime schedules the active *goroutines* (but thats not really an issue). We know that both *goroutines* are independent (there are no race conditions), so it does not matter how the execution is serialized.

The second definition is the definition of the function *concurrent*. The difference to *sequential* is that the print statement within the loop is surrounded by a typical Golang pattern. The print statement is now part of an anonymous function (with a parameter i) which should be executed as a *goroutine*. This mean calling *concurrent* results in spawning 10 *goroutines*. So this pattern is for example appropriate to spawn something like worker routines.

Channels

Channels are Golangs concept of communication. They are first class values. This means it is possible to assign *channels* to variables or use them as function parameters. Under the hood channels are typed queues that provide atomic access. The idea is that processes can write into and / or read from such a queue in order to communicate with each other. There are two general types of *channels* regarding the size of the *channel*. If the *channel* has the size zero it is said to be *unbuffered* or *synchronous*. Otherwise the *channel* is said to be *buffered* or *asynchronous*. *Unbuffered channels* are also called *synchronous* because writing into a *channel* that reached its capacity as well as reading from an empty *channel* will result in a block state. A positive side effect of this communication type is the indirect synchronization that follows from the communication relationship and the kind of used channel. Listing 2.18 demonstrate the basics of working with *channels*.

Listing 2.18: A simple example of using channels

```
1  package main
2
3  import "fmt"
4
5  func produce(c chan<- int) {
6    for i := 0; i < 10; i++ {
7      fmt.Printf("Produced: %d\n", i)
8      c <- i // synchronization
9    }
10 }
11
12 func consume(c <-chan int) {
13   for true {
14     i := <-c // synchronization
15     fmt.Printf("Consumed: %d\n", i)
16   }
17 }
18
19 func main() {
20   c := make(chan int)
21   go consume(c)
22   produce(c)
23 }
```

This example shows how channels could be used to program some kind of producer consumer relationship. In line 20 *make* is used to create a new *channel* of the type *int*. The created channel is an *unbuffered* one because no size is passed to *make*. As line 21 and 22 show the *channel* is passed as an argument to two functions. The first function is *consume*, which is defined in line 5 and the second one is *produce* which is defined in line 12. Since the channel is passed to both functions it can act as a communication path.

Produce requires a *channel* of the type *int*. One might wonder what the arrow on the right side of the keyword *chan* stand for. This simply means that the *channel* is write-only. In turn, *consume* requires an read-only *channel* (this time the arrow is on the left side). The *produce* function is very simple. Is consists of a loop that iterates 10 times. Each round it prints out the actual value of the loop counter variable and then passes the variable into the *channel c*. The arrow is used again, but this time it indicates that the variable on the right-hand is passed to the left-hand *channel*. *Consume* works quite similar but instead of writing into the *channel c* the function reads from the *channel*. The notation of reading a value from a *channel* into a variable requires additionally the assignment operator. On the first look this may seem like a break in the syntax but it is not. While *c <- i* means that the whole typed variable (not only the value) is passed to the *channel*, *'<-c'* means that a typed variable is read from the *channel*. Therefore *'<-c'* evaluates to a variable and the assignment operator is needed to copy the value of this variable into another. Let *c1* and *c2* be two *channels* of the same type and a variable should be passed from *c1* to *c2*. In such a case it is convenient to omit the assignment operator because *c2* takes the whole variable and therefore *'c2 <- c1'* is valid.

As mentioned before reading and writing into an *unbuffered channel* results in a synchronous communication. Both functions run in a separate *goroutine* (*consume* in it's own and *produce* in that one used for the main routine). Every time a read or write on the *channel* is executed the *go runtime* is called. It evaluates which of the goroutines are still blocked and which are not. After this procedure the scheduler invokes one of the non-blocked *goroutines*. This simple mechanism allows to determine the flow of execution by serializing the *goroutines* based on their communication relationships. On a single core system this would mean that *produce* and *consume* are executed alternately (and therefore concurrent) because in each case of reading and writing the actual *goroutine* will block and the scheduler picks the respective other *goroutine*. On a multi core system both functions can run partially in parallel and the read and write operation on the *channel* can be seen as rendezvous point where both functions synchronize.

The attentive reader might have noticed that *produce* is not called within a *goroutine*. The reason for this is to prevent the main from returning at an early stage — this also counts for the *sleep* at the end of Listing 2.17. If the main function returns too early the whole program will quit regardless if all *goroutines* terminated or not (especially for beginners this is a common mistake).

Listing 2.19 demonstrates a more complex example of utilizing *channels* in combination with the *select* statement.

Listing 2.19: An advanced example of using channels

```
1  package main
2
3  import "fmt"
4  import "time"
5  import "math/rand"
6
7  func produce(in chan<- int) {
8    for i := 0; i < 10; i++ {
9      fmt.Printf("Produced: %d\n", i)
10     in <- i // synchronization
11     time.Sleep(time.Duration(rand.Intn(5)) * time.Millisecond)
12   }
13 }
14
15 func consume(out []chan int) {
16   for i := 0; i < len(out); i++ {
17     go func(c <-chan int, id int) {
18       for {
19         i := <-c // synchronization
20         fmt.Printf("(%d)Consumed: %d\n", id, i)
21       }
22     }(out[i], i)
23   }
24 }
```

```
25
26 func demux(in <-chan int, out []chan int) {
27   for {
28     timeout := make(chan bool, 0)
29     go func() {
30       time.Sleep(1 * time.Millisecond)
31       timeout <- true
32     }()
33     select {
34     case i := <-in:
35       if i%2 == 0 {
36         out[0] <- i // synchronization
37       } else {
38         out[1] <- i // synchronization
39       }
40     case <-timeout:
41       fmt.Printf("Timeout\n")
42     default:
43       fmt.Println("default")
44     }
45   }
46 }
47
48 func main() {
49   in := make(chan int)
50   out := make([]chan int, 2)
51   out[0] = make(chan int, 0)
52   out[1] = make(chan int, 0)
53   go demux(in, out)
54   go consume(out)
55   produce(in)
56   time.Sleep(1)
57 }
```

At the beginning of the main function a single *channel* as well as a two dimensional *channel* array are created. The two dimensional *channel* array is a little bit more complex to instantiate because *make* is not able to determine the *capacity* and therefore can not instantiate the underlying *channels* itself.

Next to the slightly modified *producer* and *consumer* functions there is a new function which takes the name *demux*. This function is used as some kind of sorter component which has one incoming *channel* and two outgoing *channels*. The definition of *demux* begins in line 26. It consists of an unconditioned loop and a typical pattern for timeouts. A *goroutine* is used to spawn an anonymous function that justs waits a specific time and then writes into a *channel* named *timeout*. Although *timeout* is not directly passed into the anonymous function it is visible from within the function. This is possible because *golang* provides *clousures* which means that a function knows the context it was called in. The function definition is followed by a new statement namely the *select* statement. It is *golang's* lightweight version of Dijkstas's *guarded commands* as purposed in Hoare's work

CSP [Hoa78]. The major difference is that *golang* does not evaluate boolean expressions in order to decide if a command is blocked or not. Instead *golang* checks if a read or write operation on a *channel* was successful or not. This is quite similar to the *select* statement which is often used by C programmers in *IO* programming to probe read- or writeability of file-handlers. A *select* statement can consist of multiple *cases*. Each of those *cases* begin with the keyword *'case'* and has a condition that guards the code within the block that follows the *case*. In line 34 for example there is a *case* which has the condition that *channel in* is readable (has at least one variable to read). In this case the value of the read variable is assigned to the variable *i*. The guarded code is very simple and just forwards a copy of the variable *i* either to the *channel out[0]* if the value of *i* is even or to *channel out[1]* if the value of *i* is odd. The second *case* of the *select* statement is the *case* where the *channel timeout* is readable. In this example a *timeout* is not really necessary but since this is an often used pattern, it might be helpful if one already has seen a timeout construct.

If no *guards* are fulfilled then the *default case* will be executed (if it is defined). In the listing the *default case* is defined in line 42 and 43. But as long as the default *case* is defined, a *timeout* will practically never occur. This is because each loop the *timeout* gets a reset and either *i* is readable before the *timeout* happens or the *default case* is chosen. To see how the program performs with the *timeout* it is sufficient to comment out line 42 and line 43. In contrast, if the *guards* of more than one *case* are fulfilled then one of the set of possible *cases* is picked randomly (the order in which the *cases* are listed has no influence) to realize fairness between the cases.

2.3.7. The Sync Package

The *sync* [Syn] package provides synchronization primitives. These primitives are mostly used for low level libraries. High level synchronization should be done by channel and communication.
One of the most known synchronization primitives is the mutex. With the mutex it is possible to lock data structures from parallel access. Listing 2.20 shows one possible way to lock a data structure with a mutex. The type *LockableInt* inherits from the type *sync.Mutex*, this way the functions *Lock* and *Unlock* can be called on a instance of the type.

Listing 2.20: Using a mutex by inhering from the *sync.Mutex* type.

```
1 package main
2
3 import (
4     "sync"
5 )
6
7 type LockableInt struct {
8     data int
```

```
9       sync.Mutex
10  }
11
12  func main() {
13      var lockInt LockabelInt
14
15      lockInt.Lock()
16      lockInt.data = 1
17      lockInt.Unlock()
18  }
```

Another way of synchronisation is to use *WaitGroups*. A *WaitGroup* waits for a collection of *goroutines* to finish. The *WaitGroup* is initialized with a count of *goroutines* to wait for. When a *goroutine* finishes it decrements the *WaitGroup* counter and waits for the other *goroutines* to finish. If the counter reaches zero all blocked *goroutines* which are block on a wait will continue.

Listing 2.21: Using a WaitGroup

```
1   import (
2       "sync"
3   )
4
5   func main() {
6       // Allocate a new WaitGroup
7       wg := new(sync.WaitGroup)
8       for i:=0; i<10; i++ {
9           // Add a delta of 1 for every goroutine
10          wg.Add(1)
11          go DoStuff(wg)
12      }
13
14      // Wait for the goroutines to finish
15      wg.Wait()
16  }
17
18  func DoStuff(wg *sync.WaitGroup) {
19      // Call Done() on the WaitGroup when finished with the function
20      defer wg.Done()
21      // Do computioation
22  }
```

2.3.8. Gc and Gccgo

There are two different compilers available that follow the Golang specification [GoS]. The original compiler from the Golang team is called *Gc* and it is the default compiler used by the *Go* tools. The second compiler is *Gccgo* (it is a *Gcc* frontend) wrote by a community member. Each of both compilers has its pros and cons. The major differences are related to the topics *portability*, *optimization* and implementation of *goroutines*.

While *Gc* implements *goroutines* by multiplexing several *goroutines* into one OS-thread, *Gccgo* maps each *goroutine* to a single OS-thread. This means programs that spawn a lot of *goroutines* might perform better with *Gc* since thread context switching is slow compared to real *goroutine* context switching. However, with the *gold* linker it should be possible to make use of segmented stacks as well. Segmented stacks are required by the *Go runtime* to enable real *goroutines*. This means multiplexing of *goroutine* is also possible with the *gccgo* using the *gold* linker instead of *ld*. Unfortunately *gold* only supports x86 and ARM architectures.

On the other hand a drawback related to *Gc* is its general portability. Compared to *Gccgo* it is quite poor because *Gc* only supports x86 (32 bit and 64 bit) and ARM CPUs while *Gccgo* theoretically supports x86 (32 bit and 64 bit), ARM, SPARC, MIPS, PowerPC and Alpha and so on. Another benefit of *Gccgo* is that *Gcc* offers much better optimization but this in turn means the compile time is not as good as with *Gc's*.

Choosing the right compiler is therefore a question of the platform as well as of the project and how it is implemented. Since we benchmark our wrapper on a x86 system, we are able to compare both compilers. Furthermore we can test the two different linker for *gccgo* (*ld* and *gold*).

2.3.9. Cgo

Cgo is part of the *Go* tools and allows the programmer to create packages that call C code. But *Cgo* does not work with pure *Gccgo* yet. However, *Go build* allows to choose the compiler (either *gc* or *Gccgo*) and therefore allows the usage of *Cgo* together with *Gccgo*. Since *Go* tools are just available for x86 and ARM, this means that projects which choose this way has to be compiled on one of these two platforms. However, using *Gccgo* together with *Go build* enables cross compiling for other platforms, therefore using *Cgo* is not that bad.

The general idea of *Cgo* is to enrich *Go* files with special comments. These comments are recognized and evaluated. Such comments can be *#cgo* directives used to pass *cflags* to the compiler and / or *ldflags* to the linker and *//export* directives to make Go functions callable from C code. Other comments can be treated as C header informations. The *Cgo* tool is described at [Cgob] and an introduction how it should be used is given at [Cgoa].

Besides *Cgo* there is also a pure *Gccgo* solution to generate *Go* declarations from C code. Unfortunately this approach is much more complex and the *Go* source files are not compatible with those using *Cgo*. This leads to problems when building such a project with *Go build* and *Gc* which is typically the case.

3. Initial Situation & Project Goals

At the beginning of the project there only exists one other attempt to bring *MPI* to *Go* [Oth] which is far from feature complete and has data type depend functions which is not conform with the *MPI* standard.

The only other possible options to realize parallel computation over the network were to either use raw sockets or maybe use some sort of remote procedure call.
As the former sections foreshadow, Golang offers many favorable aspects that support developers in designing and writing concurrent / parallel programs. Our opinion is that especially the scientific research community can benefit from this relatively young language despite the lack of wrappers / bindings for well-established programming libraries. For this reason we would like to contribute and write a Golang wrapper for MPI.

Within the phase of development, we use OpenMPI (1.6, current stable) but since MPI is a standard and not a concrete implementation, our wrapper should also work with other implementations as well (for example MPICH2).

In order to evaluate whether the combination of Golang and MPI is suitable or not, we ported an existing MPI based benchmark tool (originally written in C) by using Golang and our function wrapper. The benchmarks have been performed on a cluster at 'Deutsches Klimarechenzentrum' (DKRZ - for specification see appendix). Comparing the observed execution times unveils the additional overhead needed by Golang (compared to the C version) and therefore we use it as the major criteria for suitability regarding performance. Detailed informations on the benchmarks are given in section 5.

4. Implementation and Realization

Writing software often requires to make fundamental decisions regarding the software and its implementation. While some general decisions can or have to be made in the initial phase of a project (design decisions) other difficulties arise during the implementation phase. In such a case it useful to dynamically explore different approaches before coming to a decision. This section can be considered as a meticulous record of the project related decisions we took and the obstacles that where encounter while implementing the wrapper. In the end we provide a small wrapper programming guide that emerged during our project time. This guide describes how a C function wrapper can be written in Golang by obeying 5 simple rules.

4.1. Design decisions at the beginning of the project

The wrapper should add as small as possible overhead to MPI while keep the feeling of programming Go instead of C. Since Cgo only supports only C and not C++ it's out of question to implement the wrapper on top of the C++ bindings. A high level interface for Go can always be implemented on top of on this C like syntax. This is further discussed in capter 6.2.

4.2. Decisions during the development phase

Now we point out the challenging parts of writing the MPI wrapper for Golang. For each challenge we give a short introduction for the specific topic before we propose concrete approaches and discuss their benefits and drawbacks.

4.2.1. Type conversions

Type conversion (also known as type casting) is a technique to change the type of an object into another type. Basically conversion is a kind of changing the interpretation and / or representation of bit data. However, manipulating bit data might result in loosing information and therefore is a potential pitfall to be aware of. For further investigations, we introduce the concept of a type system.

A Type system can be considered as a set of definitions and rules that enforce the correct usage of program components and their interaction. The purpose of these rules is to detect and avoid erroneous program code. Typically the type system of a programming language is categorized along the following three dimensions.

1. **static vs. dynamic typing:** A programming language is said to be statically typed iff the typing rules are performed during compile time. Programming languages that check the typing during execution-time are said to be dynamically typed.

2. **strong vs. weak typing:** A programming language is said to be strongly typed iff the execution of an operation is prevented in the case that the types of arguments does not fulfill the operation's specification. In contrast, a programming language may perform implicit type conversion, such a language is said to be weakly typed.

3. **safe vs. unsafe type systems:** A programming language is said to be type-safe iff it permits conversions and operations that will lead to erroneous conditions.

Golang is a statically typed language, since type checking is done during compile time. The type system does not convert indirectly and only allows performing well defined operations. This means Golang's is strongly typed. Furthermore, Golang offers (when not manually disables) bound checking for collection types, thus its type-system can also be considered as type safe.

Since the problem of loosing precision potentially comprises all types, it is a fundamental problem. Considering each type individually is impractical and unnecessary, so we will just examine the *integer* type(s) representatively and leave the reader to apply the gathered ideas to the remaining types.

In Golang all types are distinct [exceptions: (byte, uint8) and (rune, int32)] (see section 2.3). Especially for numeric types this is an important fact that means even if *int* is a 64-bit type on a particular machine, it is not possible to use arithmetic operations or comparisons mixing *int* and *int64* without using conversion [Sum12]. As indirect conversion between types may result in precision loss. Golang requires the programmer to make conversions explicitly. Dealing exclusively with Golang, conversion is relatively easy to manage.Things getting more complex if a second programming language is involved. The type *C.int* for example is distinct as well and therefrom differs from Golang's types even if its size is identical. However, another problem is that the size of *C.int* depends on the C compiler and the System. Thus, we can not make any assumptions on this type. Especially careless conversion between pointers with distinct base types implies a high risk of failures in the code, that might be hard to detect. This is why Golang also prohibits the conversion of pointers based on different underlying types. Such a procedures enforces type safe programming and allows type checking during compile time. However, most of the functions offered by MPI Implementations (written in C) require pointers [e.g. the out parameter(s)]. Fortunately Golang provides the 'unsafe' package, that allows the programmer to represent pointers of arbitrary type and hence bypass Golang's type safety (*unsafe pointer* can be considered as Golang's pendant to *void pointers* in C).

The easiest way to deal with data types (regarding the wrapper functions) would be to fall back to the C-types. This proceeding would guarantee type safety and prevent from precision loss. Additionally, the overhead caused by conversion would be avoided. While this approach might seem suitable on the first look, it implies a major drawback. The user would have to import the 'C' package and deal with the different types himself / herself. Including the 'C' package also grant access to C-bindings and therefore indirectly to an implementation of MPI that offers an interface for C. Summing up, this means that going this way would totally break the idea of writing a Golang MPI wrapper.

Another solution would be to detect the size of C.int within our GOMPI package. A proper dedicated integer type with the same size as C.int could be defined. Using this type instead of the standard int for all offered functions would allow to pass a variable to the wrapper function that performs an explicit cast without the danger to lose precision. But there is a small blemish we would like to avoid. Since all other packages do not know about this special type, the user might have to perform a lot of conversions if other packages than GOMPI are involved in the project.

At this point in time, we solve this issue by passing standard int to the function and copy the data form an int variable into a C.int variable and then pass the reference of the C.int variable into the C-function that requires the pointer. The disadvantage of this approach should be clear. Copying data introduces overhead. Sporadically copying just a single value is not that critical, but when dealing with large arrays (which is no rarity when using MPI) the copy process will waste a lot of time.

In the past an int in Golang was always represented by a 32 bit integer. In [Sum12] it is mentioned that in future the bit-representation of an int might change. This became true with Golang version 1.1 [Go1]. While with Golang 1.0.3 a C int was at least as big as Golang's int, there was no problem with precision loss. Now, this has changed. For the combination of a 32 bit C int and a 64 bit Golang int, now there is a high risk to loose precision. This could turn nasty, because the compiler has no chance to detect this kind of malicious code, which might result under special conditions in unrecognized buffer overflows.

The easiest way to solve this issue would be to change every *int* into a *int32*, but then we definitely loose the benefit for the case where both (C as well as Golang) int are 64 bit. Therefore, we currently looking for an approach that satisfies the requirement of precision loss free conversion and avoids introducing a dedicated data type.

4.2.2. Passing typed data: Arrays vs. Slices

Besides the standard types we can find in C, Golang also provides a set of high level types. For this project the important types are arrays, slices and strings, which we already introduced in section 2.3. Golang arrays are different from those one might know from C. Probably the most important difference is, that the size of an array is also part

of the type. The slice type (see section 2.3.2) is a *struct* which is generally based on the array type. It encapsulates an array by referencing its array-pointer. Two extra fields are used to store the actual size as well as the capacity. Therefore, the size of a slice is not directly part of its type. This means compared to arrays being more flexible regarding passing data of arbitrary size to a function.

The majority of functions defined by the MPI standard demand arrays of arbitrary size as input or output parameters. For this reason there is no better alternative for our wrapper functions than permanently make use of slices instead of inflexible arrays. Due to employing slices it is the logical consequence to avoid passing data sizes directly but use the slice's knowledge of its own size. Although this means to deviate from the function signatures defined by the MPI standard we consistently enforce this advance for two reasons.

1. Passing the size of an slice explicitly (redundant) to a function implies additional and avoidable overhead.

2. Formulating the wrapper functions more clear and Go-like will increase the usability and decrease error sources.

4.2.3. Passing arbitrary data: The empty interface

As mentioned before Golang is statically typed. This as well as the fact that Golang does not provide generics (see [GoG]) makes passing data of arbitrary type a bit challenging. The Problem is that MPI provide several indispensable functions that do not make assumptions of the input-parameters type (i.e. MPI_Send). Our investigations on how to solve this issue led us to two possible solutions.

The naive (and closer to the MPI standard) way, would be to pass arbitrary data via a pointer or reference to the function. In Golang there are three different types of pointers:

1. typed pointers:
   Due to fact that type checking is also performed for typed pointers, they are ineligible to solve the problem of passing arbitrary data to a specific function.

2. *unsafe.Pointer*:
   Using *unsafe.Pointer* would force the user to import the *unsafe*-package but this is something we do not want! Therefore this sort of pointer is also not appropriate to solve the problem.

3. uintptr:
   An *unitptr* would work. Using the *reflect*-package one could easily access the header of a slice an obtain the underlying *uintptr* that points to the slice's data. Therefore a wrapper function that expects a *unitptr* could indirectly receive data of arbitrary type.

But there is a more Go-like solution that is very user friendly. Instead of manually obtaining the *unitptr* and passing it to the function it would be nice to have a feature like a super-type that fits on everything. In the introduction to Golang (2.3) we mentioned the idea of interfaces. A special interface is the empty interface (*interface{}*) because every object implements this interface. Therefore *interface{}* provides a back door to pass objects of any type. Within the function it is possible to obtain the original type of the passed object. Then, using the *reflect*-package it is easy to get access to a data pointer. Our actual implementation provides a helper function for this procedure and considers the types channel, function, map, and slice.

4.2.4. MPI Definitions and how to access them from Golang space

The file *mpi.h* (shipped with the MPI implementation) contains the definitions of MPI constants and data types. While some constants and data types are directly accessible through the header file, others use preprocessor directives and are not visible for Golang. We found two approaches to make those values available for Golang.

The proposal for a first approach is quite simple. By manually tracing the required definitions in the source code of the MPI implementation, it is possible to write a particular header file for GOMPI. But this approach involve major implications. Since MPI is a standard, one should think the API (Application Programming Interface) is well defined. Unfortunately that is not true. Each MPI implementations has its own version of the header file *mpi.h*. The result is that all implementations of MPI differ more or less in some details. GOMPI has the goal to work with each MPI implementation. Thus, it would be too much work to create a dedicated header file (with the needed definitions) for each available MPI implementation. Another point is that some internal details may change in future releases of a MPI implementation (while the API remains). Therefore, it would not be enough to provide MPI implementation specific but also version dependent files that hold the definitions.

Our second approach (the one we have implemented) respect both addressed issues. A set of helper functions (defined in *scr/MPI/c-helper.h* implemented in *scr/MPI/c-helper.c*) is used to request the required values at runtime and automatically assign those values to intended variables provided by GOMPI. The involved files are:

1. src/MPI/const.go (MPI communicators, groups and handles identifier)
2. src/MPI/datatype.go (MPI Data type identifier)
3. src/MPI/operations.go (MPI Operation identifier)

4.2.5. Callback functions

Callback functions are functions that are called by other function. Such a feature is generally realized passing the callback function to a second function, which in turn execute the callback function. In Golang functions are treated as first class objects, therefore they can be directly passed as arguments to other functions. Listing 4.1 shows how callbacks could be realized in Golang.

Listing 4.1: Callback function example in Golang

```
1  package main
2
3  import "fmt"
4
5  func square(arg *int) {
6    *arg *= *arg
7  }
8
9  func apply(fn func(*int), items []int) {
10
11   for i, _ := range items {
12     fn(&items[i]) // callback
13   }
14 }
15
16 func main() {
17   items := []int{1, 2, 3, 4, 5}
18
19   callback := square
20   apply(callback, items)
21
22   for _, v := range items {
23     fmt.Printf("%d, ", v)
24   }
25   fmt.Printf("\n")
26 }
27
28 // Output: 1, 4, 9, 16, 25
```

The main purpose of callbacks is to write functions in an abstract way where the exact behavior depends on the implementation of the callback function. As one can see, the function *square* is assigned to the variable *callback*, which in turn is passed to *apply* (it is also possible to pass *square* directly to *apply*). The big advantage is, that *apply* does not need to have any information about the callback function but the address. Therefore, *square* and *apply* exhibit low coupling. The areas of application for such a technique are for example the domain of *signal handling* or *hooks*, where own functionality needs to be called by foreign code. Writing the MPI wrapper we had to implement error-handler functionality.

C can not directly access Golangs's memory. Therefore, it is not feasible to simply pass function-pointers to a C-function in order to perform callbacks on the Golang-function out of the C space. Luckily, we were able to figure out two approaches to handle this issue.

Our first approach for this problem consists in writing the error-handlers directly in C. A C-helper-function could used to pass a pointer to error-handler to the Golang space. In turn helper-functions could be used to pass data from Golang to C space and backwards. Even if this approach is applicable, it is not very handy. The user has to be familiar with C and has to provide a lot of setter and getter functions. Furthermore, mixing two different programming languages is from our point of view not a good style. It decreases the project's clarity, requires additional knowledge and is therefore a potential source of errors. For these reasons we investigated for a another, better approach that avoids writing much C code.

Before we present our current approach, we want to address another issue that occurs when the MPI error-handlers function are written in Golang.

Although variable-argument work within a pure Golang project, it does not work together with *Cgo*. Trying to call C-functions that use such argument lists, will lead to a compiler error. This issue is already known (see `http://code.google.com/p/go/issues`, issue: 975). Unfortunately some MPI functions (namely the *errhandler-functions*) require a function pointer (of a specific type) for callback functions that support variable parameter lists. As long as *Cgo* does not provide calling C function with variable-argument lists, we also can not provide this functionality in Golang.

However, to avoid loosing the whole error-handling when working with GOMPI, we provide a workaround that bypasses variable-argument lists. For each of the three error-handler types (*MPI_Comm_errhandler_function, MPI_File_errhandler_function and MPI_Win_errhandler_function*), we provide a special function declaration that fits the original function-pointer type but omits the variable-argument list (see 4.2). For each of these function declarations, we provide a special helper function that returns the particular function-pointer (see 4.3). While the function declaration as well as the related helper function reside in C space and Golang can access C space, the actual function definition / implementation of the error-handler can be written in Golang. The idea is to use the *//export* directive to make a function with a predefined name accessible for C (see 4.4). The function-pointer for a particular error-handler function could be obtained by using the related helper-function. This detour is necessary to take care that the function-pointer can be processed by C. But heads up! Declaring a function in C and using the *//export* in Golang implies that the function-name is known during compile-time. This means there can only be a limited (and constant) number of such functions. We decided to ship GOMPI with a single predefined function for each of the three error-handler types.

If more than one error-handler are needed, we advise to inspect the following files:

1. src/tests/src/errhandler.h
2. src/tests/src/errhandler.c

These files could be used as templates for declaring further functions which can be used to achieve multiple error-handlers in a GOMPI project. Both files are used to build a library that has to be linked against the project's object files. If it should not be clear how to use the declared function or the helper function, an example is provided by the file *src/tests/test-errhandler.go*.

Listing 4.2: src/tests/src/errhandler.h

```
#ifndef _GOMPIERRHANDLER_H_
#define _GOMPIERRHANDLER_H_

#include "mpi.h"

extern void GOMPI_Comm_errhandler_function( MPI_Comm *, int *);
extern void GOMPI_File_errhandler_function( MPI_File *, int *);
extern void GOMPI_Win_errhandler_function( MPI_Win *, int *);

void* getCommErrhandler();
void* getFileErrhandler();
void* getWinErrhandler();

#endif
```

Listing 4.3: src/tests/src/errhandler.c

```
#include "errhandler.h"
#include "stdio.h"

void* getCommErrhandler(){
  return &GOMPI_Comm_errhandler_function;
}

void* getFileErrhandler(){
  return &GOMPI_File_errhandler_function;
}

void* getWinErrhandler(){
  return &GOMPI_Win_errhandler_function;
}
```

Listing 4.4: Extract from src/tests/test-errhandler.go

```
1  package main
2
3  /*
4  #cgo LDFLAGS: -L/usr/lib/openmpi -L../../libs -lmpi -lcerr
5  #include <stdlib.h>
6  #include <mpi.h>
7  #include "src/errhandler.h"
8  */
9
10 ...
11
12 //export GOMPI_Comm_errhandler_function
13 func GOMPI_Comm_errhandler_function(comm *C.MPI_Comm, err *C.int) {
14   er := int(*err)
15   fmt.Printf("GO Errorcode %d\n", *err)
16
17   if er != MPI.ERR_OTHER {
18     errs++
19     fmt.Printf("Unexpected error code\n")
20   }
21
22   if MPI.Comm(*comm) != mycomm {
23     errs++
24     fmt.Printf("Unexpected communicator\n")
25   }
26
27   ...
28
29   calls++
30   return
31 }
32
33  ...
```

If error-handler functions with variable-argument lists are essential for a specific GOMPI project, we recommend to fall back to our first presented approach (there is no alternative).

4.3. How to write MPI wrapper function in Golang

For MPI implementations which are written in C, the creation of function wrappers is straight forward. Since *Cgo* creates the C pseudo package that contains ready to use C bindings, there are just few steps necessary to write the wrapper function.

1. Identify input and output parameters Golang supports multiple return values. Hence, it is a good idea to make use of this feature in order to organize the function signatures more clearly. This requires to identify and separate the input parameters form the output parameters. Exceptions are pointers to memory regions that are used as buffers for arbitrary data. As mentioned before (section 4.2.3), we use the

empty interface (instead of a pointer) to pass arbitrary data into our wrappers (a helper function is used to extract the data pointer from the interface). Especially for buffer regions this technique has proven beneficial, but it requires that also buffer which are used for output are passed into the wrapper and therefore has to be listed in the list of input parameters.

2. Eliminate unnecessary input / output parameters:
   Some MPI functions expect arrays of a specific type (typically integer or MPI_-Datatype arrays). In C those functions expect a data pointer as well as the array size. In Golang we can make use of slices instead. Since the length / size of a slice is part of its type, an extra parameter for the size is dispensable. Therefore, it is sometimes possible to reduce the parameter list in a reasonable manner to avoid additional copy overhead (call by value) and keep the signatures clear by decreasing their complexity.

3. Declare local variables for the output parameters:
   MPI functions that are realized in C just return an error code and expect for further output so called output parameters. As mentioned, we make use of the multiple return value feature of Golang. This requires that the wrapper holds a local variable for each output parameter that can be passed in the C function. Our convention is that these variables are declared using C types, so they can directly be passed into the C function and are converted to Golang types right before the wrapper returns.

4. Call the C function with in line type conversion of the input parameters:
   Before an input parameter can be passed into the C function it has to be converted to the corresponding C type. Instead of spending additional local variables which would require memory and runtime, we directly convert the input parameters within the argument list of the C function.

5. Convert the types of the local variables and return their value:
   In order to return the values the wrapper obtained via the output parameters of the C function, the local variables which hold the data have to be converted to Golang types. As befoolors that will be used to display "good", "degraded" or "bad" values using the color_good, core the conversion can happen in line.

An example of a typical and simple wrapper function (that follows the above mentioned rules) is shown in listing 4.5. The considered function is MPI_Recv(). Regarding the complexity of the wrapper it is to say that majority of our 266 wrapper functions are almost as simple as the MPI_Recv() wrapper.

Listing 4.5: The wrapper for MPI_Recv() (Extracted from src/MPI/comm.go)

```
1  func Recv(buffer interface{},
2    count int,
3    dataType Datatype,
4    source int,
5    tag int,
6    comm Comm) (Status, int) {
7
8    var status C.MPI_Status
9    bufferVoidPointer := Get_void_ptr(buffer)
10
11   err := C.MPI_Recv(bufferVoidPointer,
12     C.int(count),
13     C.MPI_Datatype(dataType),
14     C.int(source),
15     C.int(tag),
16     C.MPI_Comm(comm),
17     &status)
18
19   return Status(status), int(err)
20 }
```

4.4. Build System

When trying to build the bindings the Go build tools hit their limit limit of flexibility really fast. There where several problems when trying to build the bindings for different implementations on a single system. The solution was to ship a custom build system written in Python.

Go needs to know the location of the shared object files of MPI in order to successfully build the bindings. The Go build system is capable of utilizing *pkg-config*. But on many compute clusters the shared object files are located in some kind of custom folder and can't be found by *pkg-config*. It's possible to pass the path of the shared object files to Go by exporting it as en environment variable. The Python build script first tries to determine the path of the shared object files by using *pkg-config* and than adding custom paths that are handed to the it by the command line. Another problem is that not all MPI implementations are absolute compatible to the standard, like they introduce different error codes. Since Go doesn't support macros the source code needs to be custom generated. When installing the bindings it's necessary to pass the implementation to the build script, so it knows how to generate the source code.

Since a compute cluster can have more than one MPI implementation installed, like OpenMPI and MPICH2. It must be possible to install the bindings more than once. The build script can generate different libraries of the bindings in case there is more than one implementation present — the complete install instructions can be found at the *go-mpi* project home page on *GitHub* [go-].

5. Benchmarks

This chapter takes a look at the performance of the MPI wrapper and the Go language. It will give a brief introduction to writing fast Go code. Then the benchmark is described and the different compilers used to build it. At last the benchmark is evaluated and the results are discussed.

5.1. Writing fast Go code

When writing high performance computing applications with Go there are some aspects to take into consideration to write fast Go code. Consider a loop like Listing 5.1, the loop iterates over a slice of numbers and sums the numbers up. In addition a static offset is added in each iteration. The offset is read form the struct *attributes* and the struct is accessed in each iteration.

Listing 5.1: Summing up the numbers of a slice and adding a static offset in each iteration.

```
1 for i:=0; i < len(numbers); i++ {
2     sum = sum + numbers[i] + attributes.offset
3 }
```

At first struct lookups should be minimized as much as possible. When accessing the same struct inside a loop over and over again, the value of the struct member should be stored in a local variable. The code from Listing 5.1 can be optimised by moving the struct lookup of *attributes* out of the loop, into a variable. Listing 5.2 shows a optimised version.

Listing 5.2: Summing up numbers and adding a static offset, utilizing the range operator.

```
1 offset := attributes.offset
2 for i:=0m i < len(numbers); i++ {
3     sum = sum + numbers[i] + offset
4 }
```

Another optimization is to minimize slice lookups. For example when iterating over a slice it's faster to use the *range* operator than using a *for* loop with a counter variable. Listing 5.2 shows the listing 5.2 with *range* operator applied.

Listing 5.3: Summing up numbers and adding a static offset, utilizing the range operator.

```
1 offset := attributes.offset
2 for index, element:= range numbers {
3     sum = sum + element + offset
4 }
```

5.2. The Benchmark

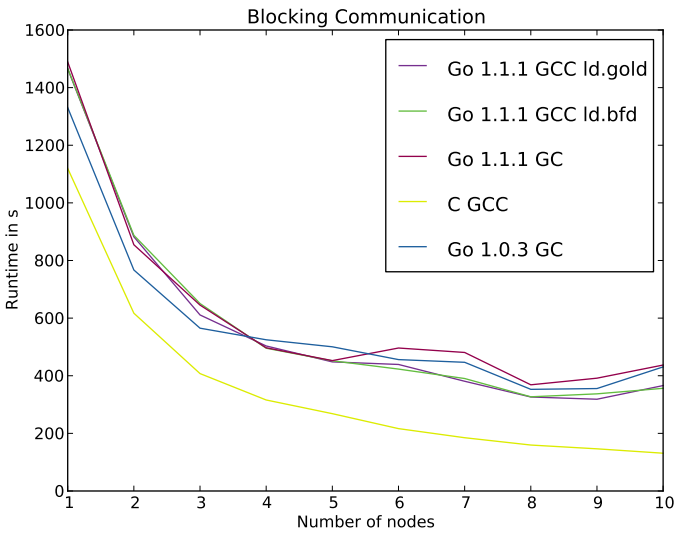
The benchmark uses the Jacobi method to solve partial differential equations. It is a line by line port of the C program die Go program is compared with. The only optimizations applied to the code a the two Go specific optimizations from Chapter 5.1. The parallelism is only archived through the MPI wrapper. Every MPI process only consists of a single *goroutine*.

5.3. Compilers

The benchmark is build with different compilers and linkers. It is compiled with the *Go GC* compiler with the 1.0.3 runtime and the 1.1.1 runtime. Furthermore the *Go GCC* compiler is used with 1.1.1 runtime. When compiling the program with *GCC* two different linker where used. One test case uses the default *GNU BFD* linker and the other uses the *gold* linker developed by Google for faster linking.

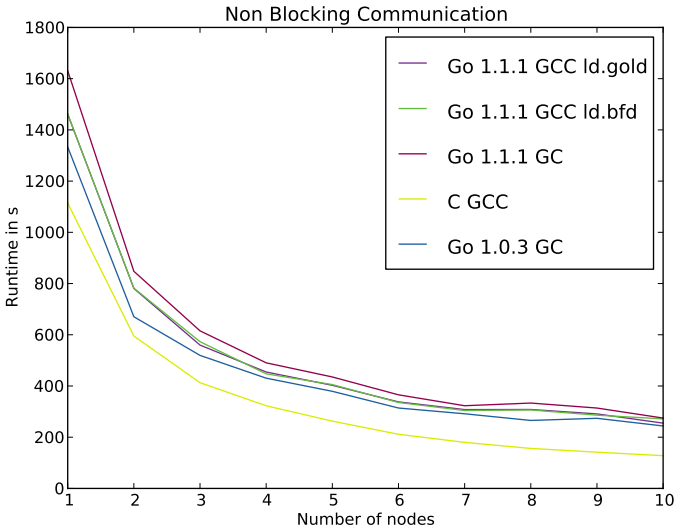
5.4. Results

Figure 5.1.: Benchmarking GO, build with different tools, versus C, using the Jacobi method to solve partial differential equation with blocking communication.



The results show that Go can not match up with the performance of C but that was to be expected. One interesting observation is that Go scales a lot nicer when using non-blocking communication when comparing the blocking C implementation with the blocking Go implementation and the and comparing the non-blocking implementations with each other.

Figure 5.2.: Benchmarking Go, build with different tools, versus C, using the Jacobi method to solve partial differential equation with non blocking communication.



Another property is that Go 1.0.3 is faster than 1.1.1. The cause of this is that Go 1.1.1 iterates slower over slices than 1.0.3 [Slo].
When looking at the non-blocking communication benchmark it is obvious that the GCC compiler dose better optimizations than the GC compiler of Go.

6. Conclusion

This project successfully implemented Go bindings for MPI 2. The source code can be found at `https://github.com/JohannWeging/go-mpi`.

6.1. Go for Scientific Computing

Go is a programming language that could have a future in scientific computing. There are several reasons for that. First it is a quit easy to learn programming language because of its slim instruction set and garbage collection (2.3). This makes Go suitable not only for computer scientists but although for all kinds for HPC users.
It's very easy to write bindings for existing C libraries for Go. See section 4.3. Common used libraries for scientific computing can be connected to Go. There are already bindings for several other libraries commonly used in scientific computing. Like the *go-hdf5*, which can be found on *Github* [HDF]. Even *GPGPU* computing is possible with Go. There are bindings for *CUDA* [CUD] and *OpenCL* [Ope], both open source and hosted on *Github*. Science Go is a quiet young language (2.3.1) it's most likely that the compiler optimizations will get better and the performance will improve over time and will get closer to the performance of C. The MPI bindings are feature complete for MPI version 2, working with OpenMPI and MPI

6.2. Future Work

6.2.1. MPI 3 and Wrapping More Implementations

Currently the implementation only supports *OpenMPI* and *MPICH2* with the features of MPI version 2. The bindings need further support of more implementations. One problem is to test the bindings with proprietary implementations of MPI because it would be necessary to obtain a license of the implementation. Maybe asking nicely helps. In addition more and more MPI implementations will support MPI version 3. The bindings will have to adapt to the new standard at some point and support the new features. Some of them where introduced in section 6.2.1. The new functions need to supported and deprecated functions need to be deleted. In general there is still a lot of testing left to ensure the correctness of the bindings. A unit test suit for each individual function has to be introduced.

6.2.2. High Level API

The *go-mpi* bindings are are very close designed to the original C API of MPI. refref But the Go language offers more features (2.3) than C which can be used to simplify the usage of MPI.

Listing 6.1: Comparing a C like MPI call against a more high level approach.

```
1 var array [100][100]int
2
3 // C like MPI call with: send buffer, count, data type, dest, tag,
     ↪ comm
4 MPI.Send(array[0][10], 10, MPI.INT, 0, 0, MPI.COMM_WORLD)
5
6 // High level call where the send method is part of a communicator
     ↪ struct
7 // with: sendbuffer, dest, tag
8 commWorld.Send(array[0][10:21], 0, 0)
```

Listing 6.1 shows a possible high level approach to MPI bindings. The *Send* function is not longer bound to the MPI package but bound to the communicator like it's done in the *C++ API* of MPI [Mes09]. However the support of the C++ bindings where discontinued in MPI 3 [MPI12] [Dis]. The count of elements to send is no longer needed because slice expressions [Go] can be used to send a portion of the array. The data type can be obtained from the data passed to the send function. This approach looks a lot nicer but it has to use reflections to obtain information about the data passed to the function. Using reflections to obtain the count of elements to send and the data type is slow. The approach adds overhead to a function call. If this high level call is executed inside a loop the overhead can be quit significant.

Listing 6.2: Incompatibility between the high level and C like *API*

```
1
2 var array [100][100]int
3 commWorld := MPI.COMM_WORLD
4
5 commWorld.Send(array[0][10:21], 0, 0)
6
7 // The new commWorld struct can used on the C like API too
8 MPI.Send(array[0][10], 10, MPI.INT, 0, 0, commWorld)
```

In addition the communicator used in the low- and the high level function is the same to ensure compatibility between the two calls. The two *APIs* should be located in different packages. One normal MPI package an a high level *HLMPI* package but they should be usable at the same time. This would allow iterative tuning the program. The program could be first programmed with the high level bindings and than use the low level C like bindings on portions of the program that often call MPI functions.

Bibliography

[Bal12] I. Balbaert. *The Way to Go: A Thorough Introduction to the Go Programming Language*. Iuniverse.Com, 3 2012.

[Cgoa] C? go? cgo! Online Resource. url: `http://golang.org/doc/articles/c_go_cgo.html` (last check: 2013-03-15).

[Cgob] Command cgo. Online Resource. url: `http://golang.org/cmd/cgo/` (last check: 2013-03-15).

[Chi12] D. Chisnal. *The Go Programming Language Phrasebook*. Developer's Library. Addison Wesley, 4 2012.

[CUD] Cuda bindings for go. Online Resource. url: `https://github.com/barnex/cuda5` (last check: 2014-02-10).

[Dis] The mpi c++ bindings: what happened, and why? Online Resource. url: `http://blogs.cisco.com/performance/the-mpi-c-bindings-what-happened-and-why/` (last check: 2014-02-10).

[FB10] R. Feike and S. Blass. *Programmierung in Google Go: Einstieg, Beispiele und professionelle Anwendung*. Open Source Library. Addison Wesley, 11 2010.

[Gcc] Gccgo, a compiler for go language. Online Resource. url: `http://golang.org/ref/spec` (last check: 2013-03-15).

[go-] go-mpi bindings. Online Resource. url: `https://github.com/johannweging/go-mpi` (last check: 2014-02-10).

[Go] The go programming language. Online Resource. url: `http://golang.org/doc/` (last check: 2013-03-15).

[Go1] Go 1.1 release notes. Online Resource. url: `http://golang.org/doc/go1.1` (last check: 2013-03-15).

[GoG] The go programming language - faq. why does go not have generic types? Online Resource. url: `http://golang.org/doc/faq#generics` (last check: 2013-03-15).

[GoS] The go programming language specification. Online Resource. url: `http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15440-f11/go/doc/gccgo_install.html` (last check: 2013-03-15).

[HDF] Online Resource. url: https://github.com/kisielk/go-hdf5 (last check: 2014-02-10).

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[MÏ1] F. Müller. *Systemprogrammierung in Google Go: Grundlagen, Skalierbarkeit, Performanz, Sicherheit*, volume 1. Dpunkt Verlag, 2 2011.

[Mes09] Message Passing Interface Forum. Mpi: A message-passing interface standard, version 2.2. Specification, September 2009.

[MPI12] MPI Forum. *MPI: A Message-Passing Interface Standard Version 3.0*, 9 2012. url: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf (last check: 2013-03-15).

[Ope] Opencl bindings for go. Online Resource. url: https://github.com/PieterD/gocl (last check: 2014-02-10).

[Oth] Mpi bindings for go. Online Resource. url: https://github.com/marcusthierfelder/mpi (last check: 2014-02-10).

[Slo] Array access time regression, significantly slower than slice. Online Resource. url: https://groups.google.com/forum/#!msg/golang-dev/TI54OYWno68/cH4xw525IF8J (last check: 2014-02-10).

[Sum12] M. Summerfield. *Programming in Go: Creating Applications for the 21st Century*. Developer's Library. Addison Wesley, 5 2012.

[Syn] Package sync. Online Resource. url: http://golang.org/pkg/sync/#WaitGroup (last check: 2013-10-02).

List of Figures

List of Tables

List of Listings

Appendices

A. Appendix

A.1. Used Software (Tools)

- Programming Languages
  - Google Go (Golang)
  - C
  - Python
- Compiler
  - GCC
  - GC
- Linker
  - GNU linker (ld)
  - Google Linker (gold)
- Libraries
  - OpenMPI
  - MPICH2
- Version Control System
  - Git `http://stud.wr.informatik.uni-hamburg.de/git/1213-gompi`
  - Github `https://github.com/JohannWeging/go-mpi`

A.2. Configuration: Testbed

- DKRZ Cluster (`cluster.wr.informatik.uni-hamburg.de`)
- 10 nodes, each:
  2 x 6 Cores (Intel Xeon Westmere 5650 @ 2.67GHz)
  12 GByte DDR3 / PC1333 RAM
  2 x Gigabit-Ethernet

A.3. Configuration: Development Systems

- Lenovo Thinkpad W530
  CPU: Ivy-bridge 4x2.6 GHz
  RAM: 8 Gigabyte
  SSD: Intel 320
  OS: Gentoo Linux
  Kernek: 3.6.11-x86_64
  GCC: 4.6.3
  Python: 2.7
  GO: 1.0.3 /1.1.1
  MPI: OpenMPI 1.6.3

- Lenovo Thinkpad X1
  CPU: Sandy-bridge 2x2,4 GHz
  RAM: 8 Gigabyte
  SSD: Crucial m4
  OS: Arch Linux
  Kernel: 3.7.11-x86_64
  GCC: 4.7.2
  GO: 1.0.3 / 1.1.1
  MPI: OpenMPI 1.6.4