GOPAR: AUTOMATIC LOOP PARALLELIZATION OF GO PROGRAMS

BY

IAN WETHERBEE

THESIS

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Engineering in the Undergraduate College of the University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Wen-Mei Hwu
Parallel computation hardware has achieved widespread consumer adoption, but software developers still need to manually exploit parallelism. Popular parallelization techniques require developers to make one or more program modifications, including source annotations, separate parallel kernel languages, manual data marshalling code, and framework-specific data containers. This additional burden forms a barrier to widespread adoption of parallel programming, and makes programs more verbose and difficult to analyze and modify. To solve these problems, this thesis introduces GoPar, an automatic loop-parallelizing compiler for the Go language that targets multicore CPUs. It aims to require no extra work from the developer to exploit parallelism and supports transforming many of Go’s language features that enable compact and expressive code. GoPar is based on a new multi-pass compiler architecture containing analysis and transformation passes for detecting parallelizable loops and outputting transformed code. GoPar removes the developer barrier to exploiting parallel hardware without sacrificing maintainable code.
To my parents, for their love and support.
# TABLE OF CONTENTS

## CHAPTER 1  INTRODUCTION ........................................ 1

## CHAPTER 2  AUTOMATIC PARALLELIZATION TECHNIQUES ........ 2

  2.1 Compile-Time ............................................. 2

  2.1.1 Data Dependence ....................................... 2

  2.2 Run-Time .................................................. 5

  2.2.1 Aliasing ............................................... 5

  2.2.2 Heuristics .............................................. 5

## CHAPTER 3  GO .................................................... 6

  3.1 Organization ............................................... 6

  3.2 Types ...................................................... 7

  3.3 Scoping .................................................... 8

  3.4 Other features ............................................. 9

    3.4.1 Short Declarations .................................. 9

    3.4.2 Multiple Assignment .................................. 9

    3.4.3 Slices .................................................. 9

    3.4.4 Control Flow .......................................... 10

    3.4.5 Defer .................................................. 11

    3.4.6 Panic, Recover ........................................ 11

    3.4.7 Concurrency ........................................... 11

## CHAPTER 4  GOPAR ............................................... 13

  4.1 Compilation Overview ...................................... 13

    4.1.1 Execution ............................................. 14

  4.2 Type Resolution ........................................... 14

  4.3 Analysis Passes ........................................... 14

    4.3.1 DefinedTypesPass ..................................... 14

    4.3.2 BasicBlockPass ....................................... 15

    4.3.3 CallGraphPass ....................................... 15

    4.3.4 AccessPass ............................................ 16

    4.3.5 AccessPropagatePass ................................ 16

    4.3.6 AccessFunctionPropagatePass ....................... 17

    4.3.7 DependencyPass ....................................... 18

    4.3.8 ParallelizePass ...................................... 18
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>Transformation Passes</td>
<td>19</td>
</tr>
<tr>
<td>4.4.1</td>
<td>RewriteLoopPass</td>
<td>19</td>
</tr>
<tr>
<td>4.4.2</td>
<td>WriteKernelPass</td>
<td>19</td>
</tr>
<tr>
<td>4.5</td>
<td>Run-Time</td>
<td>20</td>
</tr>
<tr>
<td>5.1</td>
<td>Experimental Setup</td>
<td>21</td>
</tr>
<tr>
<td>5.2</td>
<td>Results Analysis</td>
<td>22</td>
</tr>
<tr>
<td>6.1</td>
<td>Future Work</td>
<td>23</td>
</tr>
<tr>
<td>A.1</td>
<td>Stencil (7-point)</td>
<td>24</td>
</tr>
<tr>
<td>A.2</td>
<td>NBody</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>REFERENCES</td>
<td>26</td>
</tr>
</tbody>
</table>
Parallel programming is the development of programs that execute multiple threads of processing work at the same time to reduce the total amount of time the work requires. Splitting a large amount of work into chunks that can be run at the same time implies that the order of execution does not matter. Modern hardware is built to execute many tasks at the same time, since the architecture has focus shifted from increasing frequency to increasing the number of cores in a processor. Before the switch happened, developers could write a sequential program and expect it to run faster as newer hardware was released. For developers to benefit from modern hardware advances, they need to write their programs to take advantage of multiple cores.

Most algorithms begin with a sequential implementation because sequential code is easier to understand when only one instruction is executing at a time. After correctness is verified, the algorithm can then be parallelized using various methods, such as OpenMP or Intel TBB in Figure 1.1. All of these methods require the developer to modify their code in some way, which places a greater burden on future developers who have to learn an additional library before they can understand the parallel version. Instead of manually parallelizing loops, automatic parallelization enables developers to work with sequential programs that are parallelized by the compiler, and does not require modifications to the code. This makes the code easier to understand and modify later while still taking advantage of future improvements in parallel hardware architecture.

```
// OpenMP
#pragma omp parallel for
for (int i = 0; i < 10000; i++) {
    c[i] = a[i] + b[i];
}

// Intel TBB
parallel_for (0, 10000, [&](int i) {
    c[i] = a[i] + b[i];
});
```

Figure 1.1: Examples of parallel frameworks for vector addition
In order to decide whether a loop can be safely executed in parallel, a program’s outputs must be guaranteed to be the same at the end of execution as if the loop were run sequentially. Output effects include memory writes and data I/O via the disk, network, or other ports that may have side effects. Some operations that are safe to be executed out-of-order in one circumstance, such as writing to file that stores an unordered list of calculation results, cannot be proven safe for all use cases because the later usage of the file is unknown.

Conditions for safety can be analyzed to an extent during static (compile-time) analysis to disqualify loops as not being safe, but in most cases a loop cannot be declared safe without examining the values of the variables with dynamic (run-time) analysis.

2.1 Compile-Time

2.1.1 Data Dependence

Compile-time analysis of code records the read and write accesses made to variables to model the data dependence between accesses \([1, 2]\). Memory dependencies \(\delta\) are classified as true dependence \(\delta^t\) (read after write), anti dependence \(\delta^a\) (write after read), or output dependence \(\delta^o\) (write after write). It is very difficult to determine at compile-time which reads and writes form a dependence with each other in the presence of control flow, quadratic array indexing, and memory aliasing.

Array accesses can be analyzed as a function of the iteration space of the loop and the array index. Recording the accesses can be simplified by only analyzing array accesses whose index expression is linear equation containing
only loop iteration variables. If an array access does not meet this condition, reads and writes are assumed to go to every index.

\[
write(s) = \{ s \text{ is written to} \} \cup \{ v : (v \in s \land write(v)) \} \tag{2.1}
\]

\[
read(s) = \{ s \text{ is read from} \} \cup \{ v : (v \in s \land read(v)) \} \tag{2.2}
\]

The \emph{read} and \emph{write} functions return the set of variables that are read or written to in a statement.

\[
i = \langle i_1, i_2, ... \rangle \tag{2.3}
\]

The indexes of the nested loops are represented by \(i\) in (2.3), with \(i_1\) being the outermost loop.

\[
S(i) \prec S(i') \iff i < i' \tag{2.4}
\]

If each array access is a linear expression of loop indexes, each memory access \(S\) can then be represented as a function of only the iteration vector \(i\). Iteration \(i\) is executed before iteration \(i'\), so memory access \(S(i)\) precedes the access of \(S(i')\).

\[
S(i)\delta^i S'(i') \iff \text{write}(S(i)) \cap \text{read}(S'(i')) \neq \emptyset \tag{2.5}
\]

\[
S(i)\delta^a S'(i') \iff \text{read}(S(i)) \cap \text{write}(S'(i')) \neq \emptyset \tag{2.6}
\]

\[
S(i)\delta^o S'(i') \iff \text{write}(S(i)) \cap \text{write}(S'(i')) \neq \emptyset \tag{2.7}
\]

\[
S(i)\delta S'(i') \iff S(i)\delta^i S'(i') \lor S(i)\delta^a S'(i') \lor S(i)\delta^o S'(i') \tag{2.8}
\]

These equations define true (read-after-write) dependence (2.5), anti (write-after-read) dependence (2.6), and output (write-after-write) dependence (2.7). Although there are program transformation techniques for removing anti and output dependencies, for this thesis we classify them as a data dependence (2.8).

To extend the definition of data dependence to different loop iterations, all of the statements inside a loop are checked for conflict using two iteration variables \(i\) and \(i'\). To meet the requirement of 2.4, \(i < i'\).

\[
S(i)\delta S'(i') \iff S(i) \prec S(x) \prec S'(i') \land \text{write}(S(x)) = \emptyset \tag{2.9}
\]
The loop dependence between \( S(i) \delta S'(i') \) is only valid if there does not exist an iteration \( x \) that is also written to. This ensures that the dependencies are represented by iterations that are close as possible.

\[
S: \ a[i+1] = a[i] + b[i]
\]

\[
\text{read}(S) = \{a(I), b(I), I\}
\]

\[
\text{write}(S) = \{a(I + 1)\}
\]

Figure 2.1: Data dependence analysis of statement \( S \)

for \( i = 1; i < 100; i++ \) {
    S: \ a[i+1] = a[i] + b[i]
    \[1 \leq i < i' < 100\] \tag{2.10}
    \{a(i + 1)\} \cap \{a(i'), b(i'), I\} \tag{2.11}
    \{a(i), b(i), I\} \cap \{a(i' + 1)\} \tag{2.12}
    \{a(i + 1)\} \cap \{a(i' + 1)\} \tag{2.13}
}\}

Figure 2.2: Loop-carried data dependence

In Figure 2.2, the iteration space is \([1, 100]\) and the \textit{write} and \textit{read} sets are the same as in Figure 2.1. For a dependence to exist between iterations, \( i < i' \), so the values chosen for the iteration space must satisfy equation (2.10). The three equations (2.11), (2.12) and (2.13) represent the \( \delta^i, \delta^a \) and \( \delta^s \) dependencies between \( S(i) \) and \( S(i') \). There is a loop dependence \( S(i) \delta S(i') \) if there exists values of \( i \) and \( i' \) that satisfy (2.10), and one or more of the dependency set equations are not empty. In this example, the values \( i' = i + 1 \) for \( i \in [1 : 99] \) will satisfy (2.11), so there is a true dependence \( S(i) \delta^s S(i') \).

\[
\text{dependency}(L) = \begin{cases} 
\bigcup_{x=1}^{n} S_x(i) \delta S_y(i') & i < i' \\
\bigcup_{y=x}^{n} S_x(i) \delta S_y(i) & i \forall
\end{cases} \tag{2.14}
\]

This approach can be extended to multiple statements \( L = \{S_1...S_n\} \) inside a loop \( L \) by checking for loop dependence and statement dependence (2.14). Finding dependencies in this manner is called an integer linear programming algorithm, and this check is too expensive to perform for every loop. In certain cases when index expressions take on specific forms, simpler tests can be used. These include Lamport’s test [3], the Banerjee test [4], and the greatest common divisor (GCD) test [1].
2.2 Run-Time

2.2.1 Aliasing

Although many loops can be proven to have loop dependencies preventing them from being parallelized, many loops cannot be proven safe without run-time inspection. In the presence of pointers, an array $a$ may use the same underlying memory as $b$ although they use two different identifiers. Static analysis cannot detect this aliasing, so it assumes the best case scenario (no aliased variables), and requires dynamic verification of its assumption.

Other run-time methods have been proposed to ensure safety of parallel loops in the presence of non-linear array accesses, such as indirect accesses ($a[b[i]]$), complex indexing ($a[i*N+x]$), or pointer-based structures ($[*int]$). These involve checking the memory addresses contained in the inputs before launch to check if an additional dependency is introduced by two addresses being the same [5], or examining the memory accesses made inside each loop iteration to check for conflicts [6].

2.2.2 Heuristics

Even if a loop can be run in parallel, it does not necessarily result in an increase in performance. Launching iterations in parallel takes time that varies depending upon the platform. For CPUs, lightweight threads that multiplex onto operating system threads take the shortest to create and can be taken advantage of by languages such as Go (goroutines) and Python (greenlets). Creating operating system threads and context switching between them is a more expensive option used by the OpenMP and Intel TBB frameworks. Finally, parallel accelerator libraries such as CUDA/OpenCL (GPUs) or OpenMPI (clusters) induce startup and data transfer costs for every execution. Run-time checks for parallelization safety also add to the overhead of running in parallel. These range from constant time checks for aliasing between arguments to linear $O(n)$ sweeps of an array’s contents to check for duplicate indexes or pointers. As a result, heuristics as a function of loop complexity and argument sizes are created to pick between running the loop sequentially or in parallel.
Go is a general-purpose language designed with systems programming in mind. It is strongly typed and garbage-collected and has explicit support for concurrent programming. Programs are constructed from packages, whose properties allow efficient management of dependencies. The existing implementations use a traditional compile/link model to generate executable binaries. [7]

The Go language was started as an open-source project by Google in 2007 as a solution to software programming complexities created by multicore processors, networking, clusters and web servers. Go is a compiled language with first-class concurrency, garbage collection and a full standard library that focuses on programmer efficiency and productivity.

3.1 Organization

Go code is organized into packages. Each package can import other packages, and define variables, types, and functions. There are no header files, instead

package main

func main() {
    a := make([]int, 1000)
    b := make([]int, 1000)
    // populate a and b
    for i := 0; i < len(a); i++ {
        a[i] += b[i]
    }
}

Figure 3.1: A simple vector addition example in Go
any top-level definition of a type, variable or function beginning with a capital letter can be accessed by another package. Packages also cannot contain circular imports, which enforces a well-defined dependency graph.

A Go program contains one **main** package that contains the entry point `func main()`. Imported packages are resolved on the filesystem using the directories listed in the `GOPATH` environment variable.

### 3.2 Types

Types in Go are defined in reverse order from C, and identifier names come before the type. For example, `int* a[]` declares `a` as an array of integer pointers in C. In Go, the same array would be defined exactly as it is read: `a []*int`.

Structures (structs) are defined similar to C, with the added feature of being able to embed other structs anonymously (such as `*Pos` in Figure 3.2). Structs can be embedded by value (`Pos`) or by pointer (`*Pos`). Each struct contains fields that can be accessed with `instance.field` if `instance` is a struct or a pointer to a struct. The fields of any embedded structs are made available under the parent struct for direct access if there are no naming conflicts, so `instance.len == instance.Pos.len` if `instance` is a `Buffer`.

Methods can be declared on any type defined in the current package to enable object-oriented programming. Method declarations take the same syntax as a function declaration, except for an additional receiver argument, such as `(b *Buffer) in func (b *Buffer) Read(...) in Figure 3.2. New methods are declared on the type specified in the receiver argument. In Figure 3.2, the type `*Buffer` now has the method `Read`. The receiver can be the type (value receiver), or a pointer to the type (pointer receiver). Pointer receivers are passed a pointer to the type, so they can modify the type instance. Value receivers are passed a copy of the type instance, so changes made to the instance are not reflected in the calling instance. Methods on a type can be called the same way struct fields are accessed (`instance.Read(p)`). The value receiver methods of an embedded type are added to the method set of the parent struct, as well as the pointer receiver methods if the embedded type is a pointer. For example, the method `func (p *Pos) Reset()` is added to the methods available to call on the `Buffer` type because `*Pos` is
type Buffer struct {
    // embedded struct
    *Pos // {len, pos int}
    data []byte
}

func (b *Buffer) Read(p []byte) (n int, err error) {
    ...
}

func (p *Pos) Reset() {...}

Figure 3.2: Go structure and interface definition

an embedded type of Buffer.

Interfaces are implicitly satisfied at compile time if a type contains all of the methods specified in the interface, as opposed to explicitly declaring a struct to implement an interface. In Figure 3.2, Buffer implements the Reader interface because it has the Read(...) method. If one or more methods implemented to fulfill an interface definition have pointer receivers, then only pointer instances can be converted to that interface type. For example, *Buffer can be converted into a Reader interface, but Buffer cannot because the Read method requires a pointer receiver.

All types in Go are pass-by-value. However, Go’s core data structures for slices, maps and channels contain pointers to the actual data, so when they are passed as function arguments any modifications are visible to the caller. Interfaces are represented internally by a struct with two values {type_tag int, data uintptr}, so calling a function with an interface does not cause a pass by value of the underlying data. Instead, a pointer to the original data inside the interface is passed as an argument with a tag representing the underlying type of the data. In general, values will default to 0 when they are not initialized.

3.3 Scoping

Go has several levels of identifier scoping: global, package, file, and block. An identifier cannot be declared twice in the same block, but identifiers can
be declared in inner blocks that will shadow the same name in an outer block.

**Global** scope contains all built-in identifiers for types and functions made available by the Go language.

**Package** scope contains all of the constant, variable, type and function declarations made in the top level of every file in a package.

**File** scope contains the identifier of any package imports made in the file.

**Block** scope is created around `for`, `select`, `switch`, `if`, `case` and anonymous block `{}` statements, as well as function declarations. Block scope contains declarations of constants, variables, and type declarations.

### 3.4 Other features

Figure 3.3 illustrates some additional relevant language features of Go.

#### 3.4.1 Short Declarations

Go performs type inference on variable declarations if the type is left out, such as the declaration of `n`. A shorthand declaration is available as `a := x`, which means `var a = x`.

#### 3.4.2 Multiple Assignment

As shown in Figure 3.3, functions are allowed to have multiple return values. Values can be captured using a multiple-assign statement. For example, the call to `instance.Read(p)` returns two values, `n` and `err`. Variables can also be swapped safely without declaring intermediate storage variables, such as `x, y = y, x`.

#### 3.4.3 Slices

Although Go has arrays (`[100]int`), the slice type (`[]int`) is primarily used to manipulate lists. Slices are lightweight structures containing a pointer
var n = 1
var err error
p := make([]byte, 1000)
instance := &Buffer{}
def instance.Close()
for n > 0 {
    if n, err = instance.Read(p); err != nil {
        panic(err)
    }
    for i, b := range p {
        fmt.Println("Index:", i, "Value:", b)
    }
}

Figure 3.3: Examples of Go's language features

to an underlying array of data, the length of the slice, and the capacity of
the underlying array. This allows programmers to create multiple views into
a piece of data, and pass slices as arguments that can be modified inside
functions. A byte slice with an underlying array of capacity 1000 is created
with the call to make in Figure 3.3.

3.4.4 Control Flow

Go has one loop construct, the for statement. The three expression slots (ini-
tialize, condition, increment) are all optional. In Figure 3.3, the for n > 0 {} statement creates a loop that executes while n > 0. Another form of the for loop takes the keyword range and a slice, array, map, or channel argument. The range loop form returns two variables for the index of the current itera-
tion and the value at that index, i and b respectively in the range over p in Figure 3.3.

The if statement takes an additional optional initialization statement that
is executed before the if condition statement. In Figure 3.3, this is used to
first call the Read method and then check if err != nil.
3.4.5 Defer

The `defer` keyword allows a function to schedule another function to be called when the current function returns. This feature is useful to keep setup and teardown logic in the same place, as well as preventing duplication of teardown logic at each `return` site. The arguments to the `defer` function call are copied (by value) at the site of the call, not when the function returns. In Figure 3.4, the function `fp.Close()` is deferred to run at the end of the current function, regardless of how the function returns.

```go
if fp, err := os.Open("/file"); err != nil {
    return err
}
defer fp.Close()
```

Figure 3.4: Example of Go’s defer keyword

3.4.6 Panic, Recover

Go strongly discourages the use of exceptions in favor of using of multiple return values to pass along detailed errors. For handling extreme circumstances or run-time errors, the `panic` function raises an exception with an arbitrary argument. The function stops execution, calls any deferred functions, then returns to the parent and raises the `panic` again. If the panic reaches the top level, the program terminates. A `panic` can be stopped if a deferred function calls `recover`, which prevents the `panic` from continuing in the parent function. In Figure 3.5, the call to `panic(1)` causes the function to stop executing and begin executing the deferred function. Inside the deferred function, there is a call to `recover()` which returns the value 1 passed to the `panic` call. The `recover` function stops the panic from propagating to higher functions.

3.4.7 Concurrency

A key feature of Go is the ability to launch many lightweight threads, called goroutines. To launch a new goroutine, the function can be called with the
defer func() {
    if err := recover(); err != nil {
        // caught a panic(err)
    }
}()

panic(1)

Figure 3.5: Example of Go’s panic and recover

go keyword, such as go fetch(...) in Figure 3.6. The fetch function does not return any value, because goroutines cannot return values to the caller. Instead, the result is communicated through the channel result. A goroutine begins execution immediately, independent of the launching goroutine. Go encourages communication between goroutines with channels, not memory synchronization primitives. Channel-based concurrency is modeled on the principles of communicating sequential processes (CSP) [8].

```go
func fetch(chan int result, url string) {
    result <- download(url)
}

func main() {
    results := make(chan int)
    // download pages a and b concurrently
    go fetch(results, "example.com/a")
    go fetch(results, "example.com/b")
    page1 := <-results
    page2 := <-results
    fmt.Println(page1, page2)
}
```

Figure 3.6: Example of launching goroutines
Go’s performance, productivity and ease of use make it a strong contender for scientific and HPC application programming. It has the ability to give developers both the performance seen in C programs and the productivity of using Python, however developers still need to manually create concurrent data flows to take advantage of multiple cores. Parallel paradigms and lightweight libraries have been created to give developers access to the computing power, but they come at the expense of code readability.

Instead of introducing additional frameworks or encouraging constructs to launch work in parallel, we introduce GoPar, a source-to-source compiler that wraps the Go compiler. GoPar performs interprocedural dependency analysis to identify loops that are safely parallelizable and replaces them with parallel versions.

4.1 Compilation Overview

GoPar sits on top of the Go compiler to analyze and modify the program being built. Instead of running `go install nbody`, GoPar is run with `gopar install nbody`, which will output the parallelized executable `nbody`.

GoPar is influenced by the modular multi-pass LLVM compiler framework [9] by splitting up the functionality of the compiler into many small passes. Each pass can analyze the AST and make modifications to it, as well as save analysis data to be retrieved by later passes. Each pass specifies its immediate pass dependencies to be executed.
4.1.1 Execution

The main package is searched recursively for the packages that it imports using the current GOPATH. Each package is then analyzed in reverse import order through the AccessFunctionPropogatePass (Section 4.3.6). If package A imports B, then package B will be analyzed before A. After the main package has been analyzed, only the main package is run through DependencyPass (Section 4.3.7) to WriteKernelPass (Section 4.4.2). The modified main package and the run-time GoPar library are then compiled to produce the parallelized executable. This limits the search for parallelizable loops to those in the main package.

4.2 Type Resolution

GoPar resolves the type of any Go AST statement by using the identifier declaration information from AccessPass (Section 4.3.4) and DefinedTypesPass (Section 4.3.1). Resolution starts from the innermost BasicBlock outwards, then to the package level declarations, and finally the built-in global identifiers.

4.3 Analysis Passes

The majority of passes in GoPar are for analysis of the memory accesses inside each block, as well as inspection of the memory dependencies to determine if a loop is parallelizable.

4.3.1 DefinedTypesPass

The defined types pass records a map of the top-level identifiers in a package to the type of the identifier. It also attaches any defined methods to their defined types.
4.3.2 BasicBlockPass

The basic block pass creates a tree structure of all scopes in a package, as defined in Section 3.3. Each basic block represents a scope boundary that encompasses the current node and all child nodes. Each basic block maintains a reference to its parent and all child blocks. Figure 4.2 shows the basic blocks (labeled A–F) that are created.

4.3.3 CallGraphPass

The call graph pass examines every top-level function in a package and records the functions it calls, both in the current package and any imported packages. Because Go does not allow circular dependencies, and the packages are parsed in reverse import order (Section 4.1.1), every function identifier can be resolved to a function declaration. Calls to variables that store a function are also recorded, although their declarations cannot be resolved. For example, in Figure 4.2 the main function is recorded as having a call
to the *Point.Conv function, while the *Point.Conv function has calls to 

4.3.4 AccessPass

The access pass adds any identifier declarations and read or write accesses to 
identifiers to the nearest enclosing basic block. The basic block tree is ana-
lyzed in-order. The supported format for identifiers is a flat list of structure 
field accesses with optional indexing. For example, an access to a.b[i].c 
would be recorded as is. Index recording is limited to a single identifier, and 
any more complicated indexing (a.b[y*N+x].c) is recorded as an unknown 
access to the whole array (a.b).

At this stage, any calls to functions that can be resolved are replaced with 
a unique placeholder that represents all reads and writes made inside of the 
function. Any functions that cannot be resolved to an implementation with 
a defined body in Go are assumed to write to all of their arguments if the 
argument contains a pointer or is a builtin type that internally contains a 
pointer. For safety, interfaces or structures that contain pointers are not 
allowed to be parallelized by this implementation (see Section 4.3.8).

In Figure 4.2, block F is recorded as having the defined variable rSquared, 
read accesses to p.x, p.y and rSquared, and write accesses to p.t, rSquared, 
and p.r. At the location of the math.* function calls, placeholder accesses 
are inserted. Block E is recorded as having defined variable p.

4.3.5 AccessPropagatePass

The access propagate pass does a post-order traversal through the basic 
blocks to propagate up any accesses made in a scope to the parent scope. 
During each propagation, it checks if the access is an identifier defined in this 
block (recorded in Section 4.3.4), and if it is defined it does not propagate 
the access. It also checks if any of the array indexes were identifiers defined 
in this block, and if so it removes the index, leaving an unknown access to 
the array. In Figure 4.2, the accesses to p.* are propagated from block F to 
block E, while the read and write access to the local variable rSquared is not 
propagated because it is declared inside block F.
4.3.6 AccessFunctionPropagatePass

The access function propagate pass is responsible for interprocedural analysis. It analyzes the functions from the current package in called-by order using the data from the call graph (Section 4.3.3). At this point, all functions from imported packages have already been analyzed (Section 4.1.1), as well as all functions the current function calls. Any function calls that cannot be resolved have had their accesses recorded by the access pass (Section 4.3.4).

When a function call that can be resolved to a function declaration is found, the accesses made to function arguments in the top block of the function declaration are mapped to the arguments passed to the function call site of the function. Only accesses that correspond to pointer types (user pointer types or builtin slices, maps or channels) are mapped. The accesses on non-pointer parameters are ignored because they do not affect the caller’s copy of the variable (Figure 4.2). Any global accesses that are not function arguments are represented by a single access $func.Foo$. In Figure 4.2 at the call to a[i].Conv(), the accesses to p from block E are mapped to the receiver parameter a[i] because p is a pointer type *Point.

The list of mapped accesses is then propagated up through the basic blocks by inserting them at the location of the placeholder access for this function call created in 4.3.4. The same propagation logic from Section 4.3.5 is used. At this point, each basic block has a complete list of all accesses made inside of it. In Figure 4.2, the accesses that were mapped to the site of the a[i].Conv() call in block D are now propagated to block C and then B where a is defined.

![Figure 4.2: Example of propagating accesses across a function call](image)

17
4.3.7 DependencyPass

The dependency pass examines the list of read and write accesses made in each block and groups the accesses together as **ReadOnly** (only read access), **ReadWrite** (read and write accesses), or **WriteFirst** (the first access is write access). Recording **WriteFirst** dependencies will enable privatization of loop variables, however privatization is not yet implemented.

If an access is made to a subset of the memory of another access, an access will be recorded on both the subset dependency and the intersecting dependency. For example, if `a.b` is currently classified as a **ReadOnly** dependency, processing a write access to `a.b.c` will cause `a.b` to become a **ReadWrite** dependency, as well as creating a new **WriteFirst** dependency for `a.b.c`. This is because `a.b.c` is a subset of the memory of `a.b`.

4.3.8 ParallelizePass

The parallelize pass analyzes all of the loops in a package and determines if they can be safely parallelized. Only **range** loops are allowed. The dependency data in the **for** loop block is examined to ensure that the following conditions are met:

1. All reads to the array being looped over are performed through the index variable of the loop
2. No writes are allowed to variables unless they are to the index of the array being looped over
3. All variables created outside the **for** loop block must be a valid type, either:
   (a) a non-pointer type with no embedded pointer types (pointers, interfaces or slices, maps or channels)
   (b) an array or slice whose values are (a)

If all of the dependencies pass the conditions then the loop is recorded to be parallelized.
4.4 Transformation Passes

Once all of the loops have been inspected and some have been selected to be parallel, several AST modifications are needed to insert safety checks and launch the parallel loops.

4.4.1 RewriteLoopPass

The rewrite loop pass inserts a skeleton structure of empty blocks (Figure 4.3) at the location of the original \texttt{for} loop. The structure contains empty blocks to insert tests for safety, and spots for the parallel and sequential loop implementations. It also inserts an \texttt{import} statement at the top of the file to include the GoPar run-time library.

```go
__parallel := false
{
    // tests
}
if __parallel {
    // parallel
} else {
    // sequential
}
```

Figure 4.3: The AST structure for launching parallel loops

4.4.2 WriteKernelPass

The write kernel pass fills in the skeleton inserted by the rewrite loop pass (Figure 4.3). The \texttt{tests} section is filled with run-time safety checks to ensure none of the arguments are aliases of each other. The \texttt{sequential} section contains the original loop to be run if the loop does not pass the run-time safety tests. Finally, the \texttt{parallel} section contains a call to the run-time library to launch the loop body in parallel. A modified version of the loop body is inserted in a function closure that takes a single index as an argument.
This allows the scheduler to run all of the function closures in parallel using the iteration space indexes.

// original
for idx, val := range a {
    a[idx] = val + b[idx]
}

// parallel
rtlib.CPUParallel(func (_idx int) {
    idx := _idx
    val := a[_idx]
    a[idx] = val + b[idx] // original loop body
}, 0, len(a))

Figure 4.4: Transforming a sequential loop into a parallel one

4.5 Run-Time

At compile-time, GoPar assumes all arguments to a loop do not alias with each other, but this assumption must be verified dynamically before each loop is executed. Every slice argument is checked to ensure it does not overlap with any of the other arguments, resulting in $O(n!)$ aliasing checks for $n$ different slices used inside a loop.
CHAPTER 5
PERFORMANCE

5.1 Experimental Setup

Two benchmarks were used to test the automatic parallelization capabilities and performance of GoPar. The Stencil benchmark (appendix A.1) does a computation on each element of a 3D matrix involving that element’s 6 neighboring elements, with $O(n)$ complexity. The NBody benchmark (appendix A.2) that performs pairwise computation between every particle in an array, with $O(n^2)$ complexity.

The benchmarks were run on an Intel i7-3517U CPU with 2 physical cores capable of running 4 threads in parallel. Go 1.0.3 was used to compile each benchmark. The GOMAXPROCS environment variable controls how many CPU cores can be used simultaneously by the Go run-time to schedule goroutines.

![Benchmark Results](image)

<table>
<thead>
<tr>
<th>GOMAXPROCS</th>
<th>NBody</th>
<th>3D Stencil</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10.45</td>
<td>9.27</td>
</tr>
<tr>
<td>2</td>
<td>5.86</td>
<td>4.77</td>
</tr>
<tr>
<td>4</td>
<td>4.31</td>
<td>4.15</td>
</tr>
</tbody>
</table>

Figure 5.1: Benchmark results (Intel i7-3517U, 4 virtual/2 physical cores)
5.2 Results Analysis

As seen in Figure 5.1, both benchmarks were successfully parallelized to achieve a linear speedup relative to the number of cores. Loop B in the Stencil benchmark was parallelized for a 94% speedup from 1 to 2 cores, and loops A and C in the NBody benchmark were parallelized for a 78% speedup. Between 2 and 4 cores the speedup is less due to the CPU architecture only having 2 physical cores, but still achieves a 15% speedup for Stencil and a 36% speedup for NBody.
CHAPTER 6

SUMMARY

The Go language has great potential for analysis and modification by automatic tools, and the results of this study suggest that developers should write their algorithms to be optimized by compilers for specific hardware. Writing algorithms in a well-defined language without parallel-specific libraries, languages or annotations allows for easier modification and understanding by future developers. As compiler techniques continue to advance, raw algorithm code can be optimized to take advantage of future hardware innovations.

We show that GoPar is successful implementation of these philosophies for the Go language by allowing any Go code to be parallelized while maintaining compatibility with the language specification.

6.1 Future Work

Although GoPar represents a solid compiler framework for dependency analysis of Go programs and the parallelization of simple loops, there are many loops it cannot yet parallelize. Future work is aimed at expanding the analysis capabilities to find more parallel loops, and increasing performance. This includes supporting affine array accesses, arbitrary for loops, advanced aliasing analysis to allow structures containing pointers, reduction support, and heuristics for deciding if the parallel speedup outweighs the run-time setup costs.
A.1 Stencil (7-point)

```go
func stencil(system []float64, dim, advance int, change float64) (result []float64) {
    result = make([]float64, len(system))
    xdim := dim * dim
    ydim := dim
    for iter := 0; iter < advance; iter++ { // (A)
        for i, val := range result { // (B)
            x := i / xdim
            y := i / ydim
            z := i % ydim

            for j := -2; j <= 2; j++ { // (C)
                for k := -2; k <= 2; k++ { // (D)
                    for l := -2; l <= 2; l++ { // (E)
                        xj := x + j
                        yk := y + k
                        zl := z + l
                        switch {
                            case xj < 0, xj >= dim:
                            case yk < 0, yk >= dim:
                            case zl < 0, zl >= dim:
                            default:
                                val += change * system[xj*xdim+yk*ydim+zl]
                        }
                    }
                }
            }
            result[i] = val
        }
    }
    return result
}
```
A.2 NBody

```go
func (sys *System) advance(dt float64) {
    for i := range sys.results { // (A)
        body := sys.planets[i]
        for j := 0; j < len(sys.planets); j++ { // (B)
            if j == i { // don’t advance ourselves
                continue
            }
            body2 := sys.planets[j]
            dx := body.x - body2.x
            dy := body.y - body2.y
            dz := body.z - body2.z
            dSquared := dx*dx + dy*dy + dz*dz
            distance := math.Sqrt(dSquared)
            mag := dt / (dSquared * distance)
            body.vx -= dx * body2.mass * mag
            body.vy -= dy * body2.mass * mag
            body.vz -= dz * body2.mass * mag
        }
        sys.results[i] = body
    }
    for i, body := range sys.results { // (C)
        body.x += dt * body.vx
        body.y += dt * body.vy
        body.z += dt * body.vz
        sys.results[i] = body
    }
    // swap lists
    sys.planets, sys.results = sys.results, sys.planets
}
```
REFERENCES


