University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

Computer Science and Engineering: Theses, Dissertations, and Student Research | Computer Science and Engineering, Department of

12-2015

Transforming C OpenMP Programs for Verification in CIVL

Michael Rogers
University of Nebraska-Lincoln, mrogers08@gmail.com

Follow this and additional works at: http://digitalcommons.unl.edu/computerscidiss

Part of the Computer Engineering Commons, and the Programming Languages and Compilers Commons

Rogers, Michael, "Transforming C OpenMP Programs for Verification in CIVL" (2015). *Computer Science and Engineering: Theses, Dissertations, and Student Research*. 92.
http://digitalcommons.unl.edu/computerscidiss/92

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in Computer Science and Engineering: Theses, Dissertations, and Student Research by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

TRANSFORMING C OPENMP PROGRAMS FOR VERIFICATION IN CIVL

by

Michael S. Rogers

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Matthew B. Dwyer

Lincoln, Nebraska

December, 2015

TRANSFORMING C OPENMP PROGRAMS FOR VERIFICATION IN CIVL

Michael S. Rogers, M. S.

University of Nebraska, 2015

Advisers: Matthew B. Dwyer

There are numerous way to express parallelism which can make it challenging for developers to verify these programs. Many tools only target a single dialect but the Concurrency Intermediate Verification Language (CIVL) targets MPI, Pthreads, and CUDA. CIVL provides a general concurrency model that can represent programs in a variety of concurrency dialects. CIVL includes a front-end that support all of the dialects mentioned above. The back-end is a verifier that uses model checking and symbolic execution to check standard properties.

In this thesis, we have designed and implemented a transformer that will take C OpenMP programs and transform them to CIVL so that they can be verified. A large subset of the OpenMP specification is supported by the transformer. The transformer operates on a Abstract Syntax Tree (AST) representation of the program. The transformer will modify the AST to replace OpenMP constructs with equivalent CIVL constructs so that the program can be verified in CIVL. Results show that the transformer supports the most common used OpenMP constructs.

Contents

Contents iii

List of Figures viii

1 Introduction 1

1.1 The Need for Parallel Programming . . . 1

1.2 Writing Parallel Programs . . . 2

1.2.1 Parallel Memory Architectures . . . 2

1.2.2 Parallel Programming Models . . . 3

1.3 OpenMP . . . 5

1.4 Parallel Programming Challenges . . . 6

1.5 Formal Verification . . . 8

1.6 CIVL . . . 9

1.7 OpenMP Transformer . . . 10

1.8 Contribution . . . 11

1.9 Outline . . . 12

2 CIVL: Concurrency Intermediate Verification Language 13

2.1 Introduction of CIVL . . . 13

2.2 Framework . . . 14

2.3 Language 15
2.3.1 CIVL-C 15
2.3.2 Semantics 17
2.4 Verification 19
2.4.1 Commands 19
2.4.1.1 Verify 19
2.4.1.2 Show 19
2.4.1.3 Compare 20
2.4.1.4 Replay 20
2.4.1.5 Run 21
2.4.1.6 Help 21
2.4.2 Symbolic Execution 21
2.4.3 Partial Order Reduction 22
2.5 Transformation 23
2.5.1 Shared State Access 25
2.5.2 Replacing Constructs 27
2.6 Evaluation 27
2.7 Conclusion 29

3 OpenMP Transformer **30**

3.1 Approach 30
3.2 Transforming OpenMP Constructs 31
3.2.1 Parallel Pragma 32
3.2.2 For Pragma 34
3.2.3 Sections 35
3.2.4 Critical 36

3.2.5 Master . . . 37
3.2.6 Shared Read and Write . . . 38
3.2.7 Functions and Terminal Transformations . . . 39
3.3 Orphan Constructs . . . 39
3.4 Memory Model . . . 40

4 Evaluation **44**
4.1 Setup . . . 44
4.2 Results . . . 45
4.3 Failed Results . . . 46
4.3.1 Faults Caught . . . 47
4.3.2 Unimplemented Features . . . 48

5 Conclusion **49**
5.1 Limitations . . . 49
5.2 Future Work . . . 50

Bibliography **52**

A Openmp CIVL Library **56**
A.1 Support Types . . . 56
A.1.1 $omp_gteam . . . 56
A.1.2 $omp_team . . . 57
A.1.3 $omp_gshared . . . 57
A.1.4 $omp_shared . . . 57
A.1.5 $omp_work_record . . . 58
A.1.6 $omp_var_status . . . 58
A.2 Support Functions . . . 58

A.2.1 Team Creation and Destruction 58
A.2.1.1 $omp_gteam $omp_gteam_create($scope scope, int nthreads) 58
A.2.1.2 void $omp_gteam_destroy($omp_gteam gteam) . . . 58
A.2.1.3 $omp_team $omp_team_create($scope scope, $omp_gteam gteam, int tid) 59
A.2.1.4 void $omp_team_destroy($omp_team team) 59
A.2.2 Shared Variables 59
A.2.2.1 $omp_gshared $omp_gshared_create($omp_gteam, void *original) 59
A.2.2.2 void $omp_gshared_destroy($omp_gshared gshared) 59
A.2.2.3 $omp_shared $omp_shared_create($omp_team team, $omp_gshared gshared, void *local, void *status) . . 59
A.2.2.4 void $omp_shared_destroy($omp_shared shared) . . 60
A.2.2.5 void $omp_read($omp_shared shared, void *result, void *ref) 60
A.2.2.6 void $omp_write($omp_shared shared, void *ref, void *value) 60
A.2.2.7 void $omp_apply_assoc($omp_shared shared, $operation op, void *local) 60
A.2.2.8 void $omp_flush($omp_shared shared, void *ref) . . 60
A.2.2.9 void $omp_flush_all($omp_team) 60
A.2.3 Worksharing and Barriers 61
A.2.3.1 void $omp_barrier($omp_team team) 61
A.2.3.2 void $omp_barrier_and_flush($omp_team team) . . 61

A.2.3.3 $domain $omp_arrive_loop($omp_team team, int location, $domain loop_dom, $DecompositionStrategy strategy) 61
A.2.3.4 $domain(1) $omp_arrive_sections($omp_team team, int location, int numSections) 61
A.2.3.5 int $omp_arrive_single($omp_team team, int location) 61

List of Figures

1.1 Pthreads HelloWorld 5
1.2 OpenMP HelloWorld 5
1.3 OpenMP Bug 7

2.1 The CIVL framework 15
2.2 Some commonly-used CIVL-C primitives 16
2.3 A CIVL-C program with static scopes numbered; its static scope tree; and a state. 18
2.4 MPI transformation 24
2.5 CUDA transformation 25
2.6 MPI-Pthreads transformation 26
2.7 Results of running CIVL `verify` command for C programs 29

3.1 OpenMP Functions and Terminal Transformations 32
3.2 Parallel pragma transformation 34
3.3 For pragma transformation 35
3.4 Sections pragma transformation 36
3.5 Critical pragma transformation 37
3.6 Master pragma transformation 37
3.7 Shared read transformation 38

3.8 Shared write transformation 38
3.9 Orphan transformation 41
4.1 OpenMP Construct Count 45
4.2 Results of running CIVL `verify` command for C OpenMP programs . . 46
4.3 Parallel code of mxm.c 47

Chapter 1

Introduction

1.1 The Need for Parallel Programming

Parallel programming is becoming more popular as problem sizes increase. Problems can be broken up into many smaller tasks and processed simultaneously. The scale and complexity of parallel programs is increasing which makes it more difficult to ensure programs will execute as expected.

For speed increases, programmers used to rely on clock frequency increases but this rate has slowed down[1]. Programmers can no longer rely on clock frequency increases and need to find other ways to speed up program execution. Programmers have turned to other methods which include using parallel programming to fully use a processor's computing power.

Solving large problems sequentially can take an extremely long time. Solving some problems sequentially in a reasonable amount of time is not possible. When it isn't practical to wait for a sequential execution of program to finish, parallel programming can be applied to large problems to help speed up the execution. Parallel programming helps process difficult problems and large amounts of data

in a shorter amount of time.

In addition to saving time, a single CPU may not be able to fit a whole program in memory. Parallel programming can allow for larger problems to fit in memory. When the data stays in memory, a CPU can access it faster. Accessing the data faster means that the execution finishes sooner. By running a program in parallel, programmers can take advantage of worldwide resources because problems can be split up over multiple resources.

1.2 Writing Parallel Programs

There are multiple ways to express parallelism in a program. This leads to being able to break up a problem in different ways. Some problems are better suited for a specific type of parallel programming. Having multiple ways to create a parallel program allows for flexibility for the programmer.

1.2.1 Parallel Memory Architectures

There are a few main architectures for parallel programming: shared, distributed, and hybrid distributed-shared memory architectures[2].

A shared memory architecture has all of the processors access the same memory resources. All the memory is seen as global address space to the processors. This global address space is easy for programmers to understand, and sharing data is fast due to processors being able to access memory quickly. However, this solution may not always scale well due to increased memory access and the need to synchronize memory accesses.

A distributed memory architecture lets processors have their own local memory but requires a communication network to allow processors to exchange mem-

ory or data. Processors operate independently and don't have the concept of a global memory space like shared memory. This architecture allows for better scaling for processors but it can be difficult for the programmers to manage data communication across the processors and communication takes time.

A hybrid distributed-shared memory architecture combines the previous two architectures. A group of processors have access to some shared memory like in the shared memory architecture. These group of processors use a communication network to work with other groups of processors that have their own shared memory. This is a very scalable approach but it does introduce more complexity into the design.

1.2.2 Parallel Programming Models

There are multiple parallel programming models that are used. Some of the most common models are shared memory without threads, shared memory with threads, message passing, and hybrids[2].

A shared memory model has processors share an address space. The processors perform asynchronous reads and writes on the data. Mechanisms like locks can be used to control the access to the shared memory. Data is local to all of the processors so the programmer does not need to worry about communicating data between the processors.

Shared memory can also be accomplished with multiple threads. A processor can create multiple threads and split up the task among the threads. Threads share a global address space. If one thread changes some data, all other threads will also see this change. These threads need synchronization to ensure that they are not accessing the same data at the same time. A couple of popular implementations of

threading are POSIX Threads (Pthreads) and OpenMP. Pthreads is a part of the Unix operating system and is a library used in C programs.

The code in Fig. 1.1[3] shows a hello world program for Pthreads. Each of the Pthreads are created in the main method on line 15 and they go do their work in PrintHello on line 3. Pthreads requires a lot of effort from the programmer. The programmer must explicitly express the parallelism. We can see that the programmer must also exit each thread in Pthreads on line 7.

OpenMP simplifies the threading process. It is available in C, C++, and Fortran. It is based on pragmas to introduce parallelism. The code in Fig. 1.2[4] shows an OpenMP hello world program. A single parallel pragma is inserted into the code. This creates all of the threads. The body after the parallel pragma is what executes in parallel. OpenMP allows a programmer to start with sequential code and slowly increase the parallelism by adding in parallel constructs one at a time.

Threads can also be done on a GPU. A common thread model for GPUs is CUDA. CUDA takes the same shared memory for threads approach and applies it to a GPU. The processor on the GPU creates many threads and has memory that is shared among the threads.

The message passing model has processors use their own local memory instead of a global address space. Processors can exist on the same machine or across multiple machines. The processors share data by sending and receiving messages. The passing of these messages is usually implemented through library calls. The programmer has to make these library calls and is responsible for how the parallelism is implemented. The Message Passing Interface (MPI) is the standard for message passing.

Hybrid models that combine more than one model also exist. The hybrid model is a flexible model as it lets the programmer use multiple types of parallelism to

```
1 # include <pthread.h>
2 # define NUM_THHREADS 5
3 void *PrintHello(void *threadid){
4   long tid;
5   tid = (long)threadid;
6   printf(\Hello World.  I am, thread # %ld.  \n", tid);
7   pthread_exit(NULL);
8 }
9 int main(int argc, char *argv[]){
10   pthread_t threads[NUM_THREADS];
11   int rc;
12   long t;
13   for(t=0;t<NUM_THREADS;t++){
14     printf(\In main:  creating thread %ld \n", t);
15     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
16     if (rc){
17       printf(\ERROR; return code from pthread_create() is %d \n", rc);
18       exit(-1);
19     }
20   }
21   pthread_exit(NULL);
22 }
```

Figure 1.1: Pthreads HelloWorld

```
1 # include <omp.h>
2 int main(int argc, char *argv[]){
3   int nthreads, tid;
4   #pragma omp parallel private(nthreads, tid){
5     tid = omp_get_thread_num();
6     printf(\Hello World from thread = %d \n", tid);
7     if (tid == 0){
8       nthreads = omp_get_num_threads();
9       printf(\Number of threads = %d \n", nthreads);
10     }
11   }
12 }
```

Figure 1.2: OpenMP HelloWorld

break up a problem. Common hybrid models are MPI and OpenMP, MPI and Pthreads, and MPI and CUDA.

1.3 OpenMP

OpenMP is an API for shared-memory parallel programming in C, C++, and Fortran[5]. OpenMP is a pragma based language. There are pragmas for thread creation, worksharing, and synchronization. OpenMP provides constructs for

worksharing, tasking, synchronization, device, and single program multiple data. In OpenMP, the programmer explicitly specifies what parts of the program should be executed in parallel. Data in an OpenMP program can be shared or private. All of the threads can shared data but if there aren't checks in place, situations like data races may happen and unintended consequences will occur during execution.

The code in Fig. 1.3[6] shows a bug in an OpenMP program. This program will compile and run, however, the tid variable is by default a shared variable. This will cause a problem as each thread tries to use this variable. There will be a race condition involving the tid variable. Each thread will change the value of the variable and it will change the value for every other thread.The tid variable should be a private variable in the parallel section. The OpenMP specification states that OpenMP specifications "are not required to check for data dependencies, data conflicts, race conditions, or deadlocks, any of which may occur in conforming programs[5]."A program may contain some sort of fault but as long as it conforms to the OpenMP specification, OpenMP wont raise any flags which is why tools to verify programs are necessary.

1.4 Parallel Programming Challenges

Writing correct parallel code can be difficult if the problem isn't embarrassingly parallelizable. A program is embarrassingly parallelizable when there is little or no effort required to separate the problem into separate parallel tasks. A study was done showing that there is a gap of performance between basic C/C++ code and code that is parallelized and optimized by experts[7]. This gap is shown to be on average 24 times faster for parallelized and optimized code compared to basic code. This gap will only grow as computers become faster. Increasing the

```
1 # include <omp.h>
2 int main(int argc, char *argv[]){
3   int nthreads, i, tid;
4   float total;
5   #pragma omp parallel {
6     tid = omp_get_thread_num();
7     if (tid == 0) {
8       nthreads = omp_get_num_threads();
9       printf(\Number of threads = %d \n", nthreads);
10    }
11    printf(\Thread %d is starting...  \n", tid);
12    #pragma omp barrier
13    total = 0.0;
14    #pragma omp for schedule(dynamic,10)
15      for (i=0; i<1000000; i++)
16        total = total + i*1.0;
17      printf (\Thread %d is done!  Total= %e\n",tid,total);
18    }
19  }
20 }
```

Figure 1.3: OpenMP Bug

amount of parallel code in a program will help speed up the computation but it takes skilled programmers to write robust parallel code.

Writing sequential code is much easier than writing parallel code. For sequential code, a programmer is used to having an IDE check syntax, provide some static analysis, and offer debugging support. While debugging a program can be difficult, it is easier to debug a sequential program compared to a parallel program. With a sequential program, a programmer can use a debugger and get the same behavior every time because sequential programs are deterministic. Each execution will result in the same behavior. This allows a programmer to isolate problem areas in the code and diagnose the fault. In a parallel program, there are many interleavings that can happen. Many of these possibilities may result in the desired result but it only takes one possibility to produce an undesirable result for the program to fail. Each execution is not guaranteed to run in the same order as before so the same behavior is not exhibited each execution. This unpredictable behavior makes it difficult for a programmer to diagnose the fault. To debug a parallel program,

a programmer needs to think about all possibilities that can happen which is difficult to apply standard debugging methods once the magnitude of parallelism increases.

Another challenge with parallel programming is that there are many ways to express parallelism. There are libraries, language extensions, and APIs that exist. Different ways of expressing parallelism are being introduced and being modified constantly which is difficult for programmers to keep up with. It is already hard enough to be an expert in one concurrency dialect. There are hybrid parallel programs that contain more than one kind of parallelism. Programs like these introduce the possibility for concurrency errors invoking subtle interactions between concurrency dialects and models.

1.5 Formal Verification

To help close the gap between sequential and parallel code, tools are being used to provide analysis and verification of the program. When bugs or defects do happen in programs, it can be difficult to isolate and fix the bug because the bug may only occur during some runs. Some of these tools that try to find bugs use formal verification techniques.

Formal verification involves proving the correctness of a design with respect to a set of properties or requirements. It is a systematic process that uses mathematical reasoning to prove the implementation of the design is correct. Formal verification will exhaustively explore all possible input values. Approaches such as model checking and symbolic execution can be applied to help find bugs or show their absence in programs.

Model checking takes a model of a system and will exhaustively check if the

model meets the design[8]. This exhaustive check will explore all reachable states on all possible process or thread intereavings. By exploring all the possible states, it can be said if a certain property holds. After the model checking system explores all of the states, it will state that property is satisfied or that it is violated and provide a counterexample, i.e., a run of a program the program that exhibits the error.

Symbolic execution is another way to formally verify a program. Instead of concrete values, symbolic values are used as inputs[9]. These symbolic values are used to represent sets of values in the program as symbolic expressions. This means the output of the program can be expressed by the symbolic values as the input. The program can be traversed with an execution path. An execution path is series of true and false values for a series of symbolic expressions representing branch conditions. These execution paths can be combined to create a tree.

1.6 CIVL

The Concurrency Intermediate Verification Language (CIVL) framework applies model checking and symbolic execution to the problem of verifying parallel programs. CIVL provides verification of multiple concurrency dialects. The CIVL framework contains an intermediate language, CIVL-C, which can express each of the different concurrency dialects. When a concurrency dialect changes or a new one is introduced, only the front-end of CIVL will need to be changed. This front-end will parse and translate the parallel program into CIVL-C so CIVL can verify the program. CIVL allows for new analysis techniques to be implemented and be applied to a variety of the concurrency dialects.

CIVL parses the original source code using ABC, a Java-based C front-end that

preprocesses and parses C programs, to produce an abstract syntax tree (AST). An AST is an abstract syntactic representation of the source code using a tree structure. ASTs do not contain all of the information from the source program like spacing, brackets, or parentheses. Each node in the tree represents some part of the program. This node may have multiple children. For example, a program may contain a while node. This while node may contain two children: a condition and a body. The condition node would have its children form some conditional expression. The body node may contain many statement nodes which can be further broken up. ABC only recognizes programs that are written in C11.

The AST that is produced by the front-end is then modified by a series of transformers. These transformers manipulate the AST to create a pure CIVL-C representation of the program. After the AST is passed though all the transformers, it is given to a model builder to produce a model of the CIVL-C program. The verifier then takes the model and performs the verification step. CIVL checks a number of standard properties. There are also some dialect specific properties that are checked. CIVL produces a report of whether the program was correct or incorrect. If the program was incorrect then information about the error is provided by CIVL. The next chapter will provide a more detailed discussion about CIVL.

1.7 OpenMP Transformer

CIVL contains various transformers to handle different concurrent dialects. To add support for OpenMP in CIVL, we design and implement an OpenMP transformer for CIVL. Since OpenMP is mainly based on pragmas, ABC will take the source program and convert it to a CIVL-C representation. This AST representation will

contain nodes that describe the pragmas but aren't in a pure CIVL-C representation yet.

Nodes for these pragmas will be created and inserted into the AST. These pragma nodes are just an AST representation of the original pragma. The OpenMP transformer will take these pragma nodes in the AST along with any other OpenMP code and transform them into pure CIVL-C which will be used to build the model.

The transformer will work primarily with the AST. Starting from the root node, the AST is searched for OpenMP nodes or code. Based on a set of translations, the transformer takes the AST nodes and will insert, delete, and modify nodes to fit the translation. When some pragma node or OpenMP code is found, the transformer will follow the set of translations to ensure that the original semantics from the OpenMP code are still the same but expressed as CIVL-C. CIVL-C contains a number of concurrency primitives in the language to allow for this to happen. The transformer covers the majority of OpenMP pragmas and code found in the test suite in Chapter 4. Chapter 3 will provide a detailed explanation of the design and implementation of the transformer.

1.8 Contribution

In this thesis we make the following contributions:

1. We design and implement a transformer to take OpenMP C programs and transform them into a pure CIVL-C representation.
2. We perform an evaluation on various OpenMP C programs to determine how well the transformer can take programs in OpenMP C and produce CIVL-C.

1.9 Outline

The rest of this thesis is structured as follows. Chapter 2 provides the structure of the CIVL framework and how CIVL can be used to verify concurrent programs. In Chapter 3, we provide a detailed description of the OpenMP transformer design and the implementation of the transformer. Chapter 4 contains the evaluation of the transformer on a test suite of OpenMP C programs which contains a mix of verifying correct and faulty programs. In Chapter 5, we provide the conclusion along with future work.

Chapter 2

CIVL: Concurrency Intermediate Verification Language

2.1 Introduction of CIVL

There are many ways to express parallelism in computer programs. Although C is just one of many languages, there are a number of ways to expresses parallelism in C. A few very popular ways include the message passing library MPI, multi-threading library POSIX Threads, and the GPU language extension CUDA. CIVL, Concurrency Intermediate Verification Language, is a framework that creates a general model of concurrency. CIVL contains a front-end to handle different concurrency dialects. There is also a back-end verifier that takes the model produced and uses model checking and symbolic execution to check if certain properties hold. The majority of information in this chapter comes from the CIVL website[10] and accepted paper to SC15[11].

Concurrent programs are difficult to verify due to their complex and dynamic nature. Many verification tools only target one concurrency dialect. There isn't

much exchange of ideas, techniques, or codes across dialects. Many tools must reimplement the same or similar ideas for each dialect. CIVL attempts to help solve this problem. CIVL works on multiple concurrency dialects because the general concurrency model is capable of expressing multiple forms of parallelism. CIVL supports MPI, POSIX Threads, and CUDA in the C language.

2.2 Framework

The CIVL framework contains a programming language, CIVL-C, that is based on the C11. Instead of using the concurrency parts of C11, this language adds a number of concurrency primitives to the C language. Functions can be defined in any scope in CIVL-C. These concurrency primitives allows for a general concurrency model to be created that can express many different forms of parallelism. In addition to the concurrency primitives, there are additional primitives added for the verification process.

The layout of the framework can be seen in Fig. 2.1. ABC is the front-end that accepts C programs that use any of the concurrency dialects that CIVL supports or CIVL-C source code. There are transformers for each of the dialects that take the dialect specific code and create a semantically equivalent CIVL-C source code. The model builder takes the CIVL-C source and produces a CIVL model of the program. Verification techniques and static analysis can be implemented at the AST level and applied to the program that contains any of the concurrency dialects. The framework has a verifier that will produce a result of ”Yes” or ”No” with a trace.

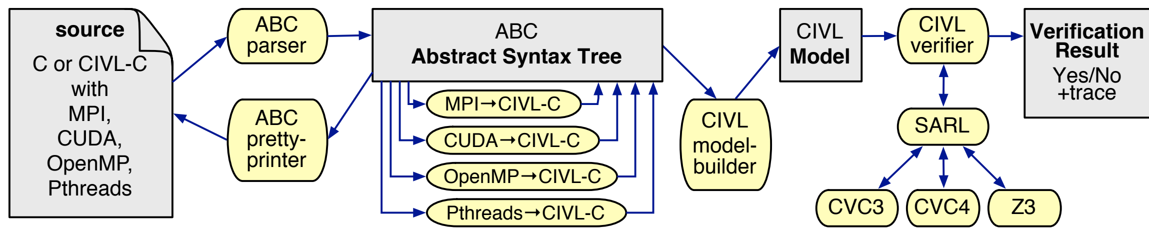

Figure 2.1: The CIVL framework

2.3 Language

2.3.1 CIVL-C

CIVL-C is based on the C11 standard of C. CIVL-C doesn't use the concurrency parts of C11 because CIVL-C has its own concurrency model. Nearly every C11 program can be expressed as a CIVL-C program. All of the CIVL-C keywords begin with a '$'. Many of the most common primitives can be seen in Fig. 2.2. CIVL-C does require that dynamically created objects are typed. This means that each malloc call must be cast to a non-void* pointer type. CIVL-C allows for function definitions to be defined in arbitrary scopes. Functions can also be spawned to create new processes. A sequential memory consistency model is used in CIVL. Every read and write to a variable happens atomically. The execution is a simple interleaving of these atomic events from the processes. To model more complex consistency models, new primitive for accesses shared variables need to be defined.

There is an implicit guard on every CIVL-C statement. This guard is a conditional statement that determines if the statement should execute. This guard is true for most statements. For $wait, the guard is enabled only when the process that the wait is for terminates. A guard can be added to any statement by using $when

`$input` : type qualifier declaring global variable to be read-only and initialized with unconstrained value of its type
`$output` : type qualifier declaring global variable to be an output, a write-only variable
`$assume(`*expr*`)` : statement informing the verifier to ignore the current execution unless *expr* holds
`$assert(`*expr*`)` : checks that *expr* holds and reports error if it does not
`$forall {`*T* `i |` *cond*`}` *expr* : universal quantification, i.e., $\forall i \in T.(cond \Rightarrow expr)$. `$exists` is similar
`$range` : type representing an ordered set of integers; e.g., `$range r1 = a .. b`
`$domain(`*n*`)` : type representing an ordered set of n-tuples of integers; includes *Cartesian* domains
`$scope` : type for reference to a dyscope; includes constants `$here` (the scope in which the expression occurs) and `$root`
`$proc` : type representing reference to an executing process; includes constant `$self`
`$malloc(`*scope*`,`*size*`)` : allocates object in heap of specified dyscope; freed with `$free`
`$for (int` *i*`,`*j*`,...:` *d*`)` *stmt* : iterates over the tuples $\langle i, j, \ldots \rangle$ in a domain *d*
`$choose_int(`*n*`)` : expression returning an integer in $[0, n-1]$, chosen nondeterministically
`$choose {` *stmt1 stmt2* `... default:` *stmt* `}` : nondeterministic selection of one enabled statement
`$spawn` *f*`(`*arg0*`, ...)` : creates and returns reference p to new process executing function f
`$wait(`*p*`)` : waits until process p terminates then removes it from the state
`$waitall(`*procs*`,` *n*`)` : like above for n processes; *procs* has type `$proc*`
`$parfor (int` *i*`,`*j*`,...:` *d*`)` *stmt* : spawns processes for each element in the domain *d* and waits until all terminate
`$when(`*guard*`)` *stmt* : guarded command; enabled only when boolean expression *guard* evaluates to *true*
`$atomic` *stmt* : executes *stmt* without interleaving of other processes

Figure 2.2: Some commonly-used CIVL-C primitives

to create the guard. Basic concurrency constructs like locks and semaphores can be created using $when.

To express nondeterministic choices one can use $choose and $choose_int. CIVL can be used to determine if two programs are functionally equivalent which

is explained later in Sec. 2.4.1.3. To help with functional equivalence, $input and $output are used as program inputs and outputs. For assume and assert statements, $assume and $assert are used.

Iteration spaces can easily represented by using the $domain type and related functions. For splitting up a for loop among a set of threads, they can be partitioned using these types and functions. A CIVL-C library function takes a function and will split it into subdomains. These threads can be run using $parfor on the domain to start all of them in their partitioned format.

CIVL-C contains types that are related to scopes and processes. These types are like scalar types in that they can be assigned to variables, passed as parameters, and returned by functions. The $spawn expression returns a new process which is of type $proc. To wait for a process, $wait is used and $proc is used as an argument for $wait. Each scope has its own heap and $malloc takes the $scope as an argument to specify which heap that the memory should be allocated in.

There are various other functions and datatypes in the CIVL-C library. This library is used to model concurrency constructs that would be inefficient or difficult to model in standard C. The library includes a barrier object $barrier for creating, joining, invoking, and destroying barriers; a communicator type $comm to insert, remove, and query messages; and a $bundle type to pack or unpack a contiguous space of memory.

2.3.2 Semantics

A CIVL-C AST is transformed into a CIVL model. This model is a lower-level representation of the program but has precise and mathematical semantics. There is a set Σ of static scopes in each model. This set of scopes makes up a rooted tree.

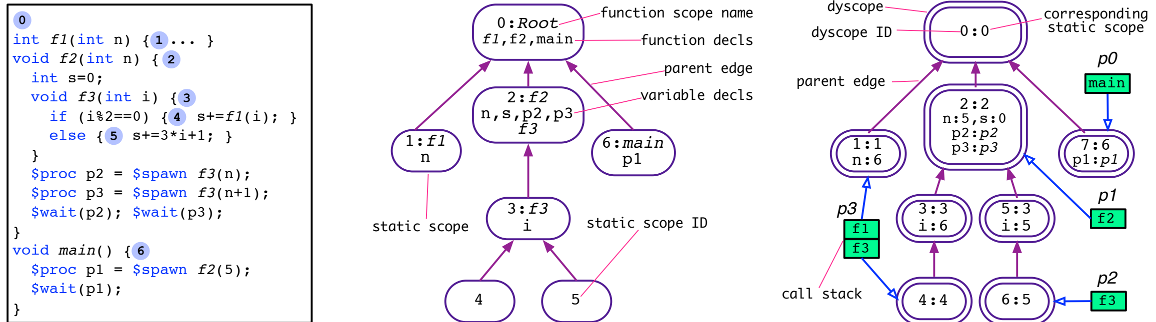

Figure 2.3: A CIVL-C program with static scopes numbered; its static scope tree; and a state.

Each element in Σ is a lexical scope in the program. Fig. 2.3 shows the original program (left), the static scopes (middle), and the tree of static scopes (right).

For each $\sigma \in \Sigma$ there is a $vars(\sigma)$. There is a heap variable in each $vars(\sigma)$. The heap variable can only be modified by $malloc and $free. There is a set of function symbols in each model. These function bodies are represented by a digraph. The nodes are locations and edges are atomic execution steps. There is a guard for each transition.

There are states in the model. Each of these states contain a set Δ of dynamic scopes which can be represented as a rooted tree. The state also has a set of processes and each process has a call stack. The call stack is a finite sequence of frames. There is a location in a program graph and dynamic scope in each frame. A dynamic scope is reachable in some state if there is some process that reaches δ in the state.

The execution starts in the initial state of the model where the state has one process with a single frame that has unknown values. At the top of the stack, the guards of the transitions are evaluated. A true one is picked and its statement is executed and the state is updated.

2.4 Verification

2.4.1 Commands

2.4.1.1 Verify

The command civl verify invokes the verification process on a program. Various options can be added to the command and filenames of the files to verify are also included in the command. The full command would be civl verify [options] filenames. This command will start with preprocessing and parsing every file. Next, all of the translation units will be merged into a single AST. Then, the transformers will translate the program to produce a pure CIVL-C representation of the program in an AST form. A CIVL model is built from the AST. Finally, the CIVL verifier is used on the model to check certain standard properties. During the verification, the verifier by default will stop at the first violation but this can be changed with the -errorBound option to search for more violations if they exist. For each violation, a description and representation of the trace are added to a log file. If two violations are of the same type and are at the same location then they are considered to be equivalent and only the one with the shorter trace is included in the log. At the end of the verification, a brief summary will be printed to the console but the log can be read to view a detailed summary.

2.4.1.2 Show

The command civl show is used to display the CIVL-C code. This command can show the code after preprocessing and parsing, after each transformation, and the final model. The AST can be printed as CIVL-C code or as hierarchical plain-text. The verifier does not run during the show command. This command is usually

used for creating CIVL-C programs and to see what the transformed program looks like.

2.4.1.3 Compare

The command civl compare takes two programs as arguments. The goal is to compare the two programs and determine if the two programs are functionally equivalent. The first program is the specification and the second program is the implementation. Each program must have corresponding $input variables. The two programs are combined to a single program and verified. Inside the combined program, each original program is in a separate function. Each program is executed and CIVL tries to assert that the outputs from each program agree. The programs are functionally equivalent if they do not violate any of the assertions. This is useful to compare sequential programs and parallel programs. They each should give the same output given the same input. This allows programmers to check that a parallel version of a program is functionally equivalent to the sequential version.

2.4.1.4 Replay

The command civl replay takes a trace from a log file and will play it back. On the replay, more or less options can be included to include more or less information during the verification. All of the transitions or states can be shown during the verification. CIVL provides references to the source code for violations that are in the trace. The original source, filename, line and column numbers, and surrounding text will be included in the violation to let the programmer know exactly where the violation is occurring.

2.4.1.5 Run

The command civl run will execute the program once. Each nondeterministic choice will be made randomly. The random seed can be specified so that this run can be reproduced. This is only a random simulation of the program.

2.4.1.6 Help

The command civl help will print out all command and options. Detailed information about all commands and options can be found on the CIVL website.

2.4.2 Symbolic Execution

The verification in CIVL uses symbolic execution. All states of the CIVL model can be explored using symbolic execution. Instead of using concrete values, symbolic expressions are used. Each state has a path condition variable which is initially set to true. A guard is checked and the new value of the path condition is updated to the conjunction of the previous path condition and the guard. If a path condition is not satisfiable then the current path can not be executed and the search backtracks to the next satisfiable path condition.

The Symbolic Algebra and Reasoning Library (SARL) is used to perform symbolic execution. SARL can create, modify, and simplify symbolic expressions. SARL can also determine if a symbolic expression is valid or not. This library is a combination of symbolic algebra and SMT theorem provers. If SARL can't solve some symbolic expression, it will use CVC3, CVC4, or Z3 depending on how CIVL is configured.

The symbolic execution performed is conservative so that if it is returned that all properties hold then these properties will hold for every execution. If a violation

is given, it is possible that it is a false positive. If SARL cannot determine the result and gives a result of unknown then a spurious report is given. This unknown result means there may be a violation. A programmer can manually check the trace to see if the unknown result is truly a violation. CIVL gives different levels of violations. CONCRETE is the highest level of certainty which means that concrete values that have been found for all inputs that satisfy some path condition cause some assertion to be violated. If a PROVABLE result is given then SARL has found a satisfiable path condition and the assertion is false. The MAYBE result means SARL has returned inconclusive results. UNKNOWN means CIVL can't handle this statement and a theorem prover was not able to produce a result.

2.4.3 Partial Order Reduction

Partial order reduction is an optimization technique that can be applied to model checking. The goal is to find a smaller number of processes where only the transitions from these processes need to be explored from some state and that if a violation exists it is still guaranteed to be found. In CIVL, there is a hierarchy of scopes. These scopes can be shared by multiple processes. Also, each dynamic scope contains a heap. This structure complicates the partial order reduction technique. The transitions used in the set of processes is known as an ample set. The dynamic scopes that the processes can reach need to be considered before determining which processes can be used to form an ample set. For the non-heap variable in each dynamic scope, follow the pointer edges to determine which objects can be reached. Now there is a set of reachable objects. If no process outside of some processes P can can reach any of the objects in the set of reachable objects then P can be used to create an ample set.

2.5 Transformation

If a program uses one of the supported concurrency dialects, it can be transformed in a CIVL-C program by using three techniques: high-level restructuring, replacing certain constructs with CIVL-C code, and implementing concurrency library functions in CIVL-C. Multiple transformers can be used on a program which allows for hybrid programs to be verified. Each transformer works on the AST and will replace or rewrite parts of the AST during the traversal of the tree. Each modification of the AST does not interfere with other transformers so many transformers can be applied. Each transformer is 1000 to 2000 lines of code. To support each transformer, the ABC grammar needs to be extended, a custom transformer is implemented, and custom support libraries written in CIVL-C are created. Transformations for MPI, CUDA, and a hybrid of MPI and Pthreads can be seen in Fig. 2.4, 2.5, and 2.6

The MPI translation creates a new main function (lines 16-19). This main function contains a $parfor that creates processes that will execute the program (lines 17-18). The _mpi_process function creates a communicator (lines 8-10) for the process. The original main function is inserted (line 12) and is then called to execute the program (line 13).

The CUDA translation creates a new main that contains a CUDA initializing function (line 29), a call to the original main function (line 30), and a CUDA finalizing function (line 31). The _gen_main function (lines 24-27) is the original main function. Each CUDA kernel (lines 6-23) has various functions and calls that are added. The block that each thread operates on is computed (lines 11-15). The block is processed in parallel by calling $cuda_run_procs() (line 16) and the grid is processed in parallel by calling the same function (line 19).

```
1 ⟨external-definitions⟩
2 int main(void) { ...  }
```

Original MPI Code

transformed to CIVL-C

```
1 $input int _mpi_nprocs;
2 $input int _mpi_nprocs_lo,_mpi_nprocs_hi;
3 $assume(_mpi_nprocs_lo <= _mpi_nprocs &&
4     _mpi_nprocs <= _mpi_nprocs_hi);
5 $mpi_gcomm _mpi_gcomm =
6     $mpi_gcomm_create($here, _mpi_nprocs);
7 void _mpi_process(int _mpi_rank) {
8   MPI_Comm MPI_COMM_WORLD =
9       $mpi_comm_create($here, _mpi_gcomm,
10      _mpi_rank);
11  ⟨external-definitions⟩
12  int _gen_main(void) { ... }
13  _gen_main();
14  $mpi_comm_destroy(MPI_COMM_WORLD);
15 }
16 void main() {
17   $parfor (int i : 0 .. _mpi_nprocs-1)
18     _mpi_process(i);
19   $mpi_gcomm_destroy(_mpi_gcomm);
20 }
```

Figure 2.4: MPI transformation

In the MPI and Pthreads hybrid translation, _mpi_process is created and called the same way as the regular MPI translation. The _gen_main function (lines 24-27) creates and runs each thread. The Pthreads transformation provides a global (lines 6-7) and thread local variable (lines 19-20) for accessing the thread pool.

There is a support library for each concurrency dialect. This library is written in CIVL-C and defines types, constants, and functions that are to be used by the transformed code. All primitives and functions for each dialect have special prefixes in each library to help separate different transformations. In addition to creating new primitives, many of the primitives described earlier in Sec. 2.3 can be used in these support libraries. These support libraries are meant to be a replacement of the dialect specific libraries like mpi.h. Transformers can introduce new variables or functions to help with the transformation. Each of the new variables or functions introduced have their own dialect specific prefix.

```
1 __ global__void add(int *a, int *b, int *c) {
2   int tid = blockIdx.x;
3   if (tid < N) c[tid] = a[tid]+b[tid];
4 }
5 int main(void) {
6   ...
7   add<<<gridDim,blockDim,0,stream>>>(a, b, c);
8   ...
9 }
```

Original CUDA Code

transformed to CIVL-C

```
1  void $cuda_run_procs(dim3 dim, void process(uint3)) {
2    $domain(3) dom = ($domain){0 .. dim.x-1,
3      0 .. dim.y-1, 0 .. dim.z-1};
4    $parfor(int x,y,z :  dom) process((uint3){x, y, z});
5  }
6  void _cuda_add(dim3 gridDim,dim3 blockDim,
7      size_t _cuda_mem_size, cudaStream_t _cuda_stream,
8      int *a,int *b,int *c) {
9    void _cuda_kernel($cuda_kernel_instance_t
10     *_cuda_this, cudaEvent_t _cuda_event) {
11     void _cuda_block(uint3 blockIdx) {
12       void _cuda_thread(uint3 threadIdx) {
13         int tid = blockIdx.x;
14         if (tid<N) c[tid]=a[tid]+b[tid];
15       }
16       $cuda_run_procs(blockDim, _cuda_thread);
17     }
18     $cuda_wait_in_queue(_cuda_this, _cuda_event);
19     $cuda_run_procs(gridDim, _cuda_block);
20     $cuda_kernel_finish(_cuda_this);
21   }
22   $cuda_enqueue_kernel(_cuda_stream, _cuda_kernel);
23 }
24 int _gen_main(void) {
25   ... _cuda_add(blocksPerGrid, threadsPerBlock, 0,
26     stream, a, b, c); ...
27 }
28 int main(void) {
29   $cuda_init();
30   _gen_main();
31   $cuda_finalize();
32 }
```

Figure 2.5: CUDA transformation

2.5.1 Shared State Access

All concurrency dialects have some shared state. Examples are that MPI contains buffered messages and Pthreads contains shared variables. In message passing there is sending and receiving of messages. Threading has reads and writes to shared variables. The access to this shared state needs to be controlled to ensure

```
1 ⟨external-definitions⟩
2 void* run(void *arg) { ... }
3 int main(void) {... pthread_create(...,run,...); ...}
```

Original MPI-PThreads Code

transformed to CIVL-C

```
1 ...
2 void _mpi_process(int _mpi_rank) {
3   MPI_Comm MPI_COMM_WORLD =
4       $mpi_comm_create($here, _mpi_gcomm, _mpi_rank);
5   // Pthread library definitions ...
6   $pthread_gpool_t $pthread_gpool =
7       $pthread_gpool_create($here);
8   int pthread_create(pthread_t *thread, ...,
9       void *(*start)(void*),void *arg) {
10    $atomic{
11      thread->thr = $spawn start(arg); ...
12      $pthread_gpool_add($pthread_gpool, thread);
13    }
14    return 0;
15  }
16  // ...  more Pthread library definitions ...
17  ⟨external-definitions⟩
18  void* run(void *arg) {
19    $pthread_pool_t _pthread_pool =
20        $pthread_pool_create($here, $pthread_gpool);
21    ...
22    $pthread_exit((void*)0, _pthread_pool);
23  }
24  int _gen_main(void) {
25    ... pthread_create(..., run, ...); ...
26    $pthread_exit_main((void*)0);
27  }
28  ... _gen_main(); ...
29 }
```

Figure 2.6: MPI-Pthreads transformation

that the program behaves as expected. This shared state creates a combinatorial explosions during model checking. CIVL allows developers to include knowledge about independence of library operations. This allows developers to improve on already existing techniques. CIVL developers can encode library dialect specific information in a Java interface called an Enabler.

For example, there is a global communicator in MPI. This is on the heap of the shared scope. The global communicator brings together the message queues and metadata about the state of processes within the communicator. This global communicator is visible to all processes. There is a local communicator for

each local scope. Using the local communicator, the library gets send or receive operations but these operations can't be accessed by another process since it doesn't have access to that local communicator.

A similar pattern and ideas are used in other transformers. The Pthreads transformer has a global and local handle for accessing threads in the thread pool. The local handle can only access threads that are able to be accessed from that scope. CUDA has a threading hierarchy that is expressed in nested functions in CIVL that are executed in parallel.

2.5.2 Replacing Constructs

For each concurrency dialect, there are specific constructs. Some of these constructs can be handled by restructuring or using CIVL-C libraries but sometimes the constructs need to be replaced with CIVL-C code that is created by the transformer. These constructs first need to be identified. For a multi-threaded concurrency dialect, the original constructs need to be rewritten in CIVL-C to allow for shared variables across threads and private variables for each thread. Using the CIVL-C construct $parfor, threads can be created. The transformer will take the existing code and replace it with the appropriate CIVL-C code to keep the program semantics the same. The only change will be that it is a CIVL-C representation.

2.6 Evaluation

CIVL developers gathered a number of programs for the concurrency dialects supported by CIVL. These programs cover a large subset of the constructs in each concurrency dialect. Examples were mainly chosen from user communities or previous analysis efforts. Currently, there are efforts to support large case studies in

CIVL. Some programs were slightly modified to accept inputs to modify the scale of the program, number of threads, or number of processors. These modifications were made to speed up execution times since many of these examples were run on personal machines or small servers. Also, some assertions were added to programs to ensure the results were correct.

CIVL developers gathered C programs that use a variety of concurrency dialect from a variety of sources. By gathering a wide range or programs, the goal is to cover a large subset of the constructs that appear in each dialect. Examples were picked from the user communities, e.g. LLNL tutorial exercises, or from previous analysis efforts, e.g. the SV-COMP Pthreads benchmark. The 24 examples reported in this section comprise of 2954 source lines of code.

The programs were kept in their original state but with a few exceptions, small modifications were made to to support command line parameters to specify the problem scale. The scale includes matrix size, number of time steps in simulation, number of processes, and number of threads. Also, some assertions were added to programs. Intermediate and/or final results may be checked during numerical computations to make sure they agree with the results of sequential versions.

Fig. 2.7 has 24 examples in it that were used during the evaluation. The "Type" column indicates the concurrent dialect: C=CUDA, M=MPI, P=Pthreads. Two letter codes are hybrid programs. Each program has a "+"/"-" for a positive or negative verification result under the "R" column. The lines of code are shown under the "LoC" column. The total number of states and transitions for the verification are given in the "States" and "Transitions" columns. The "Time" column is the time is rounded to the nearest second. The total amount of memory in megabytes is given in the "Memory" column. The number of valid calls, "ValidCalls" column, and the number of calls, "Prove" column, there required an external prover are shown.

Type	Example	R	LoC	States	Transitions	Time	Mem	ValidCalls	Prove	Scale
M	diffusion1d.c [12]	+	164	118272	117440	30	1120	463080	146	1≤NX,NSTEPS≤5,1≤NP≤3
M	diffusion2d.c [12]	+	274	489379	485418	247	1859	2473368	49	1≤NX,NY,NSTEPS≤5,NPX=NPY=2
M	mpi_prime.c [13]	+	105	28281	28276	14	1385	79342	382	{PRIMES}⊆[10, 15],1≤NP≤4
M	mpi_pi_send.c [13]	+	120	112922	112357	101	732	305872	4241	1≤DARTS,ROUNDS,NP≤2
M	sum_array.c [12]	+	72	81852	81366	13	1393	439555	89	1≤NX≤20,1≤NP≤5
M	wave1d.c [12]	+	194	98091	97216	54	897	420943	240	1≤NX,NSTEPS≤5,1≤NP ≤3
M	wave1dBad.c [12]	-	192	496	495	4	650	2118	46	1≤NX,NSTEPS≤5,1≤NP≤3
M	gausselim.c [12]	+	293	408185	406073	115	628	1769614	1911	1≤NROW≤4,1≤NCOL≤2,1≤NP≤3
M	matmat_mw.c [12]	+	104	85982	85294	18	1315	646793	36	1≤M,N,L≤3,1≤NP≤4
C	cuda-omp.cu [14]	+	99	9401	10331	8	1409	142221	92	1≤NBLK≤4,1≤NTperBLK≤2
C	dot.cu [15]	+	99	13713	13921	6	650	110745	73	1≤N≤6, 1≤NTperBLK≤4
C	mm.cu [16]	-	146	877	878	3	515	3632	15	NBLK=4,NTperBLK=1
C	sum.cu [17]	+	72	1297	1314	3	515	15679	6	NBLK=4,NTperBLK=2
C	vectorAdd.cu [18]	+	148	4796	5179	5	650	69055	15	1≤N≤6, 1≤NTperBLK≤4
P	bug4.c [19]	-	85	12162	12597	4	650	37915	3	NT=3, ITR=5, THD=7, NSTEPS=10
P	queue_ok_longest_...c [20]	+	126	68364	71574	16	843	121365	2	SIZE=800
P	read_write_lock_...c [20]	-	38	1758	1878	2	515	1762	0	NT=4
P	sync01_true...c [20]	+	42	320	329	1	515	602	0	NT=2
P	03_incdec_true...c [20]	+	56	448	453	1	515	116	3	NT=3
MP	mpithreads_both.c [19]	+	87	32908	35778	10	1384	92395	5	NP=2, NT=2, VLEN=5
MP	MP-infinity-norm.c [21]	+	146	2861	2896	6	650	6228	39	NP=NT=2, 1≤NROWS, NCOLS≤3
MP	MP-matrix-vector.c [21]	-	170	7151	7905	7	921	25693	30	NP=3, 1≤NT=NROWS=NCOLS≤4
MP	MP-pie-collective.c [21]	+	79	23006	24723	12	1381	61623	44	NP=3, 1≤NITR≤5, NT=NITR/NP
MP	anl_hybrid.c [22]	-	43	27118	26932	10	1385	121099	4	NP=NT=2

Figure 2.7: Results of running CIVL `verify` command for C programs

Lastly, the scale is given that was used during program execution in the "Scale" column. These executions were on an Appie iMac running OSX 10.9.2 (64 bit) with a 3.5 Ghz Intel i7 processor.

CIVL found 8 unintended errors in programs that were thought to be correct. Upon further inspection, CIVL developers found that each error reported was a real defect.

2.7 Conclusion

CIVL supports a number of concurrency dialects and can support more dialects with additional transformers. Programs were scaled down so they they could be verified in a few minutes but CIVL can handle larger scale sizes with more time. CIVL is novel in its support for multiple concurrency dialects. In the next chapter, an implementation of a OpenMP transformer in CIVL is presented. This transformer will allow for C OpenMP programs to be verified in CIVL.

Chapter 3

OpenMP Transformer

3.1 Approach

To support OpenMP, three components are needed: an extension of the ABC grammar, a custom transformer, and custom support libraries. This section focuses mainly on the implementation of the custom transformer. Code was added to ABC to parse each of the OpenMP pragmas and functions. These nodes contain all of the same information as the source code but represented in an AST.

The primary goal of the transformer is to take AST nodes that represent the pragmas in OpenMP and transform them into an AST that is a pure CIVL-C representation. The transformer modifies, adds, and deletes nodes in the AST. Any change to the AST can be modeled by a set of rules.

The OpenMP transformer starts at the root node of the program and performs a depth first search of the tree trying to match nodes that need to be modified. If a node is found that needs to be changed, the changes are made to the AST and then the traversal of the tree continues. If the node that is being inspected does not match any OpenMP construct, the traversal continues with the children of

the node. If a node does not have any children then the traversal backtracks as expected in a depth first search.

When the nodes are being inspected to see if they match some OpenMP construct, the node's type is being checked. The OpenMP constructs that are replaced with CIVL-C code are: `parallel` pragmas, `for` pragmas, `sections` pragmas, `critical` pragmas, `master` pragmas, `atomic` pragmas, `barrier` pragmas, `single` pragmas, OpenMP functions, and shared reads and writes. All of these constructs contain some code or pragma that is not compliant with CIVL-C and must be transformed into pure CIVL-C code.

3.2 Transforming OpenMP Constructs

This section shows the construct that is matched during the transformation. The new CIVL-C representation is given to replace the original code. Fig. 3.1 shows all of the terminal rules and functions that are used in subsequent transformation rules. Rules that are all capital letters are terminal rules. Some of these functions or rules define variables which can be used later in translations. Lines that are underlined are to be transformed recursively. The transform() method is used to show that the body needs to be transformed recursively and any OpenMP constructs will be replaced during the traversal of this AST nodes.

All definitions of functions and variables that begin with omp_ can be found in Appendix A. These functions and variables make up the custom support library for OpenMP in CIVL. The library is created to allow CIVL to create certain functions and variables that can be used many times and would be difficult to implement in the transformer.

Original	Transformed
	OpenMP Functions
omp_get_num_threads() omp_get_num_procs() omp_get_max_threads() omp_set_num_threads(n)	_omp_nthreads 1 _omp_thread_max _omp_nthreads = n
	Terminal Statements

THREADS(N)	$elaborate(N);
	int threads = $choose_int(N);
	int _omp_nthreads = 1 + threads;

RANGE(X, Y)	$range _omp_thread_range = X .. Y - 1;
DOMAIN(X)	$domain(1) _omp_dom = ($domain){_omp_thread_range};
ARRIVE_LOOP_DOMAIN(X)	$domain _omp_my_iters = $omp_arrive_loop(_omp_team, 0, ($domain)X, 2);
ARRIVE_SECS_DOMAIN(X)	$domain(1) _omp_my_secs = $omp_arrive_sections(_omp_team, 0, X);
CREATE_GTEAM	$omp_gteam _omp_gteam = $omp_gteam_create($here, _omp_nthreads);
CREATE_GSHARED(S)	$omp_gshared _omp_S_gshared = $omp_gshared_create(_omp_gteam, &(S));
CREATE_TEAM	$omp_team _omp_team = $omp_team_create($here, _omp_gteam, _omp_tid);
DEFINE_LOCAL_STATUS(S)	TYPE(S) _omp_S_local; TYPE(int) _omp_S_status;
DEFINE_PRIVATE(P)	TYPE(P) _omp_P_private;
CREATE_SHARED(S)	$omp_shared _omp_S_shared = $omp_shared_create(_omp_team, _omp_S_gshared, &(_omp_S_local), &(_omp_S_status));
BARRIER_FLUSH	$omp_barrier_and_flush(_omp_team);
DESTROY_SHARED(S)	$omp_shared_destroy(_omp_S_shared);
DESTROY_TEAM	$omp_team_destroy(_omp_team);
DESTROY_GSHARED(S)	$omp_gshared_destroy(_omp_S_gshared);
DESTROY_GTEAM	$omp_gteam_destroy(_omp_gteam);

Figure 3.1: OpenMP Functions and Terminal Transformations

3.2.1 Parallel Pragma

The `parallel` pragma is a fundamental construct that starts the parallel execution. When a thread reaches a `parallel` construct, a team of threads is created. These threads will execute the `parallel` region together. The thread to encounter the `parallel` construct is the master thread in the `parallel` region. All of the threads that are created, including the master, execute the `parallel` region. The team of threads will execute for the whole duration of the `parallel` region.

In CIVL, `$parfor` has the same syntax as `$for`. In `$parfor`, one process is spawned for each element of the domain. Each process has local variables corresponding to the iteration variables. Each process executes the body of the `$parfor`.

Each of the processes wait at the end for the other processes. There is a barrier at the end of the loop and the spawned processes are destroyed at the end of the loop.

The transformed CIVL-C code will declare the number of threads as seen in Fig. 3.2 (line 2). This produces a variable called `_omp_nthreads` which will be used to create the range. The range and domain are determined (lines 3-4). The range is an expression that represents the range from the lower bound to the higher bound. The definition of range is given in Fig. 3.1 along with all other all capitalized rules. The domain takes expressions of type $range like the range variable just created. A `$domain` variable created represents the domain of dimension which is the Cartesian product of the ranges. The domain variable created is called `_omp_dom` and is used later in the `$parfor`.

Then the global team and gshared variables are created (lines 5-6). The '*'means that there may be 0 or more of that statement added. The global team contains all of the information for threads and shared variables inside the parallel region. The global shared objects are global objects that every thread can see.

The `$parfor` is the parallel loop statement (line 7). An iteration variable, `_omp_tid` is created for each partition. Then for each partition of the domain in the `$parfor`, there is a team (line 8), local and status variables for each shared variable (line 9), private variables (line 10), and shared objects for each shared variable (line 11). The body is transformed recursively to change any other OpenMP code to pure CIVL-C. A barrier and flush (line 15) occurs after the `$parfor` and all shared objects and the team are destroyed (lines 16-17). Finally, the global shared objects and the global team are destroyed (lines 19-20).

```
1 # pragma omp parallel private(P) shared(S) num_threads(N)
2   body
3 }
```
Parallel Pragma

transformed to CIVL-C
```
 1 {
 2   THREADS(N)
 3   RANGE(0, _omp_nthreads)
 4   DOMAIN(_omp_thread_range)
 5   CREATE_GTEAM
 6   CREATE_GSHARED(S)*
 7   $parfor(int _omp_tid:  _omp_dom){
 8     CREATE_TEAM
 9     DEFINE_LOCAL_STATUS(S)*
10     DEFINE_PRIVATE(P)*
11     CREATE_SHARED(S)*
12     {
13       transform(body)
14     }
15     BARRIER_FLUSH
16     DESTROY_SHARED(S)*
17     DESTROY_TEAM
18   }
19   DESTROY_GSHARED(S)*
20   DESTROY_GTEAM
21 }
```

Figure 3.2: Parallel pragma transformation

3.2.2 For Pragma

The `for` pragma is a loop construct that has iterations of one or more loops associated with it. This construct is always inside a parallel construct so these loops are executed in parallel by the threads in the team. The iterations are distributed among all of the threads.

The for pragma can operate on one or multiple for loops. The `range` and `domain` for the loop are determined (lines 2-3) in Fig. 3.3. The `range` comes from the for loop(s) associated with the for pragma. In the case of the transformation here, the for loop is from 0 to N so that is used in the range variable. The `domain` takes expressions of type `$range` like the range variable just created. A `collapse` clause can be added to the for loop. If the `collapse` clause is used then multiple for loops can be associated with the for pragma. All of the available threads can

```
1 # pragma omp for
2 for(i=0; i<N; i++)
3   body
4 }
```
For Pragma

transformed to CIVL-C
```
1 {
2   RANGE(0, N)
3   DOMAIN(_omp_range)
4   ARRIVE_LOOP_DOMAIN(_omp_loop_domain)
5   $for(int i:  _omp_my_iters){
6     transform(body)
7   }
8   BARRIER_FLUSH
9 }
```

Figure 3.3: For pragma transformation

be partitioned across all of the loops that are associated by the `collapse` clause. Then multiple ranges would be created and used as the arguments for the domain variable. The `nowait` clause can be used to eliminate the barrier at the end of the transformation. The elimination of this barrier lets each thread exit the for loop and continue execution without waiting for all threads to finish the loop.

Another domain variable is created by the arrive loop function (line 4). The domain from the arrive loop function returns a subset of the original domain. This domain will determine which iterations each thread will execute. The $for iterates over the loop and the body is transformed (lines 6-8). After the loop, a barrier is applied (line 9).

3.2.3 Sections

The `sections` pragma is a non-iterative worksharing construct. The `sections` construct contains structured blocks that have the label `section`. Each `section` is executed by a single thread in the team. The `sections` construct must be inside of a `parallel` construct. Only the threads in the `parallel` region can work in the `sections` region.

```
1 # pragma omp sections
2 # pragma omp section
3   body0
4 # pragma omp section
5   body1
6 ...
```

Sections Pragma

transformed to CIVL-C

```
1 {
2   ARRIVE_SECS
3   $for(int i :  _omp_my_secs){
4     switch(i){
5     case 0:{
6       transform(body0)
7       break
8     }
9     case 1:{
10      transform(body1)
11      break
12    }
13    ...
14    }
15  }
16  BARRIER_FLUSH
17 }
```

Figure 3.4: Sections pragma transformation

The sections construct consists of one sections pragma and at least one section pragma. The translation begins with getting a domain from the arrive sections function (line 2) in Fig. 3.4. A subset of the domain is returned by the arrive sections function. This domain will determine which thread will execute each `section`. The `_omp_my_secs` specifies which of the sections each thread will execute. A `$for` is use to loop through the threads to execute each section. There is a switch statement that contains a case for each section (lines 5-12). Each case contains the transformed body from each section pragma. There is a barrier at the end of the translation (line 16).

3.2.4 Critical

The critical construct takes a block and restricts the access to a single thread at a time. The critical pragma can have the name of the lock associated with the

```
1 # pragma omp critical(a)
2   BODY
```

Critical Pragma

---

transformed to CIVL-C
Earlier in the program add a global declaration for the boolean variable for the critical section.

_Bool _critical_a=$false

```
1 {
2   $when(!_critical_a) _critical_a=$true
3   transform(body)
4   _critical_a=$false
5 }
```

Figure 3.5: Critical pragma transformation

```
1 # pragma omp master
2   body
```

Master Pragma

---

transformed to CIVL-C

```
1 {
2   if(_tid==0){
3   transform(body)
4 }
```

Figure 3.6: Master pragma transformation

pragma. If there is a name, then a global using that name is created for that critical lock. If there is no name, then a generic lock name is used for the critical lock. The transformation in Fig. 3.5 checks when the lock is not acquired and will set the lock to true when it acquires it (line 2). The body of the critical section is transformed (line 3) and the lock is then released (line 4).

3.2.5 Master

The master pragma is for a block that only the master thread will execute. In Fig. 3.6 if the thread id is zero (line 2) then it is the master thread and only it can execute the block (line 3).

```
1 EXPR //With shared variable VAR
```

Shared Read

transformed to CIVL-C

```
1 {
2   TYPE tmp;
3   $omp_read(_omp_VAR_shared, &(tmp), &(_omp_VAR_local));
4   replaceVar(EXPR); //Replace shared variable name with temporary variable name
5 }
```

Figure 3.7: Shared read transformation

```
1 TYPE VAR = EXPR //VAR is shared variable
```

Shared Write

transformed to CIVL-C

```
1 {
2   TYPE tmp;
3   tmp = EXPR;
4   $omp_write(_omp_VAR_shared, &(_omp_VAR_local), &(tmp));
5 }
```

Figure 3.8: Shared write transformation

3.2.6 Shared Read and Write

OpenMP has shared variables across threads. The reads and writes to these variables need to be controlled. The shared read is an expression that contains a variable that has a shared variable that is read from in some expression. In Fig. 3.7, a temporary variable is created (line 2) and it has the same type as the shared variable. A CIVL function in the custom OpenMP support library is used to read the variable from the shared variable and store it in the temporary variable (line 3). The shared variable name is then replaced with the temporary variable name in the expression (line 4).

Writes to shared variables must also be controlled. In Fig. 3.8, a temporary variable is created and it has the same type as the shared variable (line 2). The temporary variable is assigned the value of the right had side of the statement (line 3). A CIVL function in the custom OpenMP support library assigns the shared variable to the temporary variable (line 4).

3.2.7 Functions and Terminal Transformations

The functions that are part of Fig. 3.1 show OpenMP functions and their direct translation. OpenMP contains functions to get and set the number of threads and processes. These appear more often than any other OpenMP function. The terminal transformations are labels that were given in rules above.

3.3 Orphan Constructs

In OpenMP, orphan constructs are possible. An orphan construct is when there is a region whose binding thread set is the current team, but it is not nested within another construct that started the binding region. For example, a `parallel` region may be created and then some function is called in the `parallel` region. The function that is called may contain a `for` construct that executes the threads in parallel over the loop.

In the translation of some of the constructs, there are certain variables like the local and status variables and thread id. These variables were defined in the `parallel` construct which are in a different scope and are thus not available in the orphan construct. An example of an orphan program can be see in Fig. 3.9. This example has the `main` that contains the `parallel` pragma which contains a call to `dotprod()`. The `dotprod()` contains a `for` pragma which is not in the same scope as the `parallel` pragma. The `for` pragma is inserted into the scope of the `parallel` pragma before the call to the function at lines 22-27 of the transformed code. The original function has all OpenMP constructs taken out of it as seen on lines 7-10 of the transformed code.

This transformation is done so that the transformed code from the `for` pragma has access to variables created by the `parallel` pragma. CIVL allows nested

functions so this transformation is valid. Functions are inserted inline in the scope of the `parallel` pragma which causes code bloat. As orphan are deeper down calls, more functions need to be inserted.

3.4 Memory Model

OpenMP has a weak consistency memory model. This model requires there to be explicit management of the global and local memory views. There are flush operations to ensure that the memory views stay consistent. Each thread can have a temporary view of the memory. Each thread has access to the global shared memory but also has its own thread private memory.

There are two kinds of variables in a parallel section: shared and private. For each shared variable, the variable becomes a reference to the original variable. For each private variable, a new copy of the original variable is created. If multiple threads try to write to the same memory without being synchronized, a data race can occur. If some thread reads from a memory location that some other thread has written to without synchronization, a data race can also occur. Reads and writes to shared variables must be controlled. New functions and variables are introduced in CIVL-C and used in the OpenMP transformer to ensure that access to shared variables is properly controlled.

Flush operations are used to provide a guarantee of consistency between a thread's temporary view and memory. A flush allows that a value written by one thread to be read by another thread. First, thread one writes a value to the variable. Then the variable is flushed by thread one which is followed by the variable being flushed by thread two. Finally, the value of the variable is read by thread two.

The transformed CIVL-C code manages sets of threads that are grouped into

```
1 #define VECLEN 100
2 float a[VECLEN], b[VECLEN], sum;
3 float dotprod (){
4   int i,tid;
5   tid = omp_get_thread_num();
6   #pragma omp for reduction(+:sum)
7     for (i=0; i < VECLEN; i++){
8       sum = sum + (a[i]*b[i]);
9       printf(\tid= %d i=%d\n",tid,i);
10    }
11 }
12 int main (int argc, char *argv[]) {
13   int i;
14   for (i=0; i < VECLEN; i++)
15     a[i] = b[i] = 1.0 * i;
16   sum = 0.0;
17   #pragma omp parallel
18     dotprod();
19   printf(\Sum = %f\n",sum);
20 }
```

Orphan construct

---

transformed to nested

```
1 #define VECLEN 100
2 float a[VECLEN], b[VECLEN], sum;
3 float dotprod(){
4   int i;
5   int tid;
6   tid = 0;
7   for(i = 0; i < 100; i++){
8     sum = sum + ((a[i]) * (b[i]));
9     printf(\tid= %d i=%d\n",tid,i);
10  }
11 }
12 int _gen_main(int argc, char* argv[]){
13   int i;
14   for(i = 0; i < 100; i++)
15     a[i] = b[i] = 1.0 * i;
16   sum = 0.0;
17   #pragma omp parallel default(shared){
18     float dotprod(){
19     int i;
20     int tid;
21     tid = omp_get_thread_num();
22     #pragma omp for reduction(+:  sum)
23       for(i = 0; i < 100; i++){
24         sum = sum + ((a[i]) * (b[i]));
25         printf(\tid= %d i=%d\n",tid,i);
26       }
27     }
28     dotprod();
29   }
30   printf(\Sum = %f\n",sum);
31 }
32 int main(){
33   char* _gen_argv_tmp[10];
34   for(int i = 0; i < 10; i++)
35     _gen_argv_tmp[i] = &(_gen_argv[i][0]);
36   _gen_main(_gen_argc, &(_gen_argv_tmp[0]));
37 }
```

Figure 3.9: Orphan transformation

teams. The $parfor construct is used to fork and join a set of threads. For each shared variable, there are sets of variables created to provide the data views necessary to model the OpenMP memory model. For some variable X, _omp_X_local is created to provide a thread local view of the shared variable. Each thread also creates _omp_X_status that will record which threads have accessed the variable since the last flush of the variable.
The status variable can have three different values:

- 0=EMPTY: Local is empty
- 1=FULL: Local is occupied and no writes have been made to it
- 2=MODIFIED: Local is occupied and writes have been made to it

For each shared variable, the local variable has the shared variable's value and the status is FULL.

Each thread's view of the thread is coordinated by _omp_X_shared which is coordinated by _omp_X_gshared. By keeping a record of this data about shared variables, CIVL can determine when shared variable accesses exhibit unsafe and undefined behavior.

In CIVL, a function void $omp_read($omp_shared shared, void *result, void *ref) is used to read a shared object. In the function, ref is a pointer to the copy of the shared variable and the result is the temporary variable that the read value is stored in. In CIVL, the read of a shared variable starts with checking if the status value is EMPTY. The shared data is copied into the local copy. Then the data held by the local copy is read and returned.

The function void $omp_write($omp_shared shared, void *ref, void *value) is called by a thread to write to a shared object. In the function, ref is a pointer to the

local copy of the shared variable and value is what is being written to the local copy of the shared variable. In CIVL, the write of a shared variable takes the local copy of the shared variable and writes a value to it. Then the status value is set to MODIFIED.

The barrier and flush operation is implemented by the function void $omp_barrier_and_flush($omp_team team). This performs a barrier and all flush on all shared objects that are associated with the team. During the flush, the operation depends on the value of the status variable.

- EMPTY: no op
- FULL: The status is changed to EMPTY and the local copy is set to the default value
- MODIFIED: The local copy is copied to the shared copy and the status is set to EMPTY. The local copy is set to the default value.

Scheduling the threads for parallel loops is a difficult challenge for efficient verification. If a loop has n iterations and there are k threads, there can be k^n different schedules. The iteration domain abstractions make it possible for all loop schedules to be explored.

Chapter 4

Evaluation

4.1 Setup

CIVL can support C programs written in various concurrency dialects. CIVL is able to support C programs that use MPI, Pthreads, and CUDA. With the addition of this transformer, CIVL is able to provide support for C OpenMP programs.

A set of C programs were gathered from a variety of sources. The goal was to cover a large subset of all of the OpenMP constructs. Examples were taken from user communities, previous analysis efforts, and code available on repositories such as GitHub and Bitbucket. There are 48 programs in the CIVL OpenMP examples directory with 5411 source lines of code. The count of each of the constructs transformed can be seen in Fig. 4.1.

As with the previous tests performed within CIVL, very few modifications were performed to the programs. Modifications were made to support command line parameters to determine the problem scale, number of steps in simulations, or number of threads. Also modifications added assertions to some examples that did not already contain them. These assertions are used in places like when a

Construct	Count
parallel	116
parallel for	123
parallel sections	4
for	99
sections	10
master	22
critical	40
omp_get_num_threads()	84
omp_get_num_procs()	10
omp_get_max_threads()	6
omp_set_num_threads(n)	12

Figure 4.1: OpenMP Construct Count

program performs a numerical computation to ensure that the result is correct.

4.2 Results

Fig. 4.2 presents data on 21 examples from our evaluation. The tests target the full space schedules. Each "Example" has a name and a citation for the source of the example. A positive ("+") or negative ("-") is shown in the "R" column. The number of source lines of code, "LoC", and the number of "States" and "Transitions" explored by the verifier are reported. The "Time" column is the time is rounded to the nearest second. The total amount of memory in megabytes is given in the "Memory" column. The number of valid calls, "ValidCalls" column, and the number of calls, "Prove" column, that required an external prover are shown. Lastly, the scale is given that was used during program execution in the "Scale" column.

All of the tests were executed on release 1.5 of CIVL on an Apple MacBook Pro

Example	R	LoC	States	Transitions	Time	Mem	ValidCalls	Prove	Scale
canonicalForLoops.c	+	40	20543	22067	14.21	332	48952	7	1≤NT≤2
dijkstra_openmp.c[23]	-	227	N/A	N/A	N/A	N/A	N/A	N/A	NV=6,1≤NT≤2
dotProduct1.c[6]	+	18	7177	7315	15.71	229	28832	7	N=8,1≤NT≤2
dotProduct_critical.c[6]	+	33	38135	38886	16.35	381	65398	7	N=10,1≤NT≤2
dotProduct_orphan.c[6]	+	25	70026	70161	38.45	400	267632	7	N=100,1≤NT≤2
heated_plate_openmp.c[23]	+	156	399797	405708	89.71	422	882242	13	M,N=5,EPSILON=0.1,1≤NT≤2
matProduct1.c[6]	+	61	647938	647938	125.9	459	1018179	7	NRA,NRB,NRC=5,1≤NT≤2
matProduct2.c[6]	-	105	8859	8853	17.16	139	9148	7	NRA,NRB,NRC=10,1≤NT≤2
md_openmp.c[23]	-	281	11679	11676	97.46	416	25991	606	ND=1NP=10,NSTEPS=10,1≤NT≤2
mxm.c	-	103	127533	127528	38.74	414	278073	8	l,m,n=10,1≤NT≤2
pi.c	+	69	40816	40964	27.66	391	166446	10	N=100,1≤NT≤2
poisson_openmp.c[23]	-	269	1655	1653	14.11	239	2950	9	NX,NY=10,1≤NT≤2
quad_openmp.c[23]	+	86	60109	60326	28.09	424	225237	9	N=100,1≤NT≤2
omp_bug5.c[6]	-	54	29208	29275	14.92	274	35313	7	N=10,1≤NT≤2
omp_bug5fix.c[6]	-	54	26345	26367	13.19	325	35390	7	N=10,1≤NT≤2
omp_bug5fixfix.c	+	54	108343	108941	21.38	414	147044	10	N=10,1≤NT≤2
prime_openmp.c[23]	+	77	761027	762875	116.87	453	2656643	15	n_hi=500,1≤NT≤2
random_openmp.c[23]	+	82	395121	400941	62.3	412	490290	4	N=100,1≤NT≤2
raceCond1.c	+	11	86476	87450	18.64	390	115646	4	A=50,1≤NT≤2
satisfy_openmp.c[23]	+	131	40972	41295	23.02	352	172964	7	N=5,1≤NT≤2
sgefa_openmp.c[23]	-	668	46750	47369	37.08	409	129802	62	N=10,100,1000,1≤NT≤2

Figure 4.2: Results of running CIVL `verify` command for C OpenMP programs

running OSX 10.9.5 (64 bit) with a 2.4 Ghz Intel Core 2 Duo processor with 4 GB of 1067MHz DDR3 of memory. CIVL was configured to use Z3 4.3.2, CVC3 2.4.1, and CVC4 1.4.

The data indicates the breadth of OpenMP programs that the transformer can support. Most of the programs were scaled down so that they could be verified in a few minutes but CIVL is sufficiently scalable that parameters can be set to higher values. The examples that were verified show that CIVL is capable of verifying OpenMP programs that contain the most common constructs.

4.3 Failed Results

Each test result has a positive or negative results. A positive result means that all of the standard properties hold. A negative result means that there is some violation in the program and a trace is provided to help identify the fault. Some of the faults are true problems in the program. Other violations that are found are due to unimplemented features in CIVL or the OpenMP transformer.

```
1 # pragma omp parallel shared ( a, b, c, l, m, n ) private ( i, j, k )
2 # pragma omp for
3   for ( j = 0; j < n; j++){
4     for ( i = 0; i < l; i++ ){
5       a[i+j*l] = 0.0;
6       for ( k = 0; k < m; k++ ){
7         a[i+j*l] = a[i+j*l] + b[i+k*l] * c[k+j*m];
8       }
9     }
10  }
```

Figure 4.3: Parallel code of mxm.c

4.3.1 Faults Caught

In Fig. 4.2, a violation was found in the mxm.c example. The parallel part of the program can be seen in Fig. 4.3.The shared variable a is accessed by $a[i + j * l]$ in 3 nested for loops (line 7). The variable l is defined to be 10 at the start of the program. The loops that use i and j range from 0 to nd which allows the i and j variables to have the values 10 and 0, respectively. Also, i and j can have the values 0 and 1, respectively. This would let two threads access a[10] at the same. This results in a race condition and a violation is given. The sgefa_openmp is like the mxm.c example. There are two for loops, one inside another, that iterate over i and j. There is a write to $y[i + j * n]$ which for when $n = 10$, the pairs $i = 10, j = 0$ and $i = 0, j = 1$ will result in the same index which is a race condition.

The omp_bug5.c example is provided by LLNL in an OpenMP tutorial. This example was engineered to have a deadlock in the example which CIVL finds. The provided fixed example, omp_bug5fix.c resolves the deadlock condition but the loop iteration variable i is not declared to be a private variable so it is implicitly determined to be shared. Having this variable as shared lets a race condition happen with respect to i. By adding i to the private clause, omp_bugfixfix.c is able to be verified as a correct program.

The poisson_openmp.c example contains a sqrt() on line 307 and CIVL finds

that the argument for the function may be less than zero. For the sort function, the argument must be positive. The md_openmp.c example fails due to a divide by zero error. The potential and kinetic variables are added up and then used in a denominator of a fraction. This value is found to be zero in some case which is a violation that CIVL finds.

4.3.2 Unimplemented Features

The dijkstra_openmp example contains incompatible types. The types in the translation match up and the bug is in the OpenMP support library. The matProdcut2.c example attempts to dereference a null pointer. The transformed code appears to be correct and the bug is in some CIVL support library.

Chapter 5

Conclusion

5.1 Limitations

Many of the common OpenMP constructs and functions are covered but not all of them are. SIMD, tasking, device, and cancellation constructs are not covered by the transformer. While these are not widely used constructs in OpenMP, support for these constructs will broaden support for OpenMP C programs. Only some of the OpenMP functions are covered. Some of the execution environment and lock routines are not currently supported. OpenMP timing routines are not supported by the transformer. For some of the current supported constructs some of the clauses are not fully supported.

During verification of some of the programs, the state space expands very quickly slowing down the execution in CIVL. As better and more effective techniques are applied during the verification, the transformer may not properly transform a program for the verifier to analyze the program in the best possible way. The transformer will need to evolve as the CIVL framework changes.

Orphan constructs are handled as described in Section 3.3 have some conse-

quences. The program size can increase rapidly as more functions are inserted into certain scopes. The functions that are added into the parallel scope can make it difficult to read and understand the program which may make it hard for a developer to maintain the code.

CIVL supports programs written in C11 and it is a strict adherence to the language. Some elements in the source code that are not compliant with the C11 standard may fail in CIVL. CIVL supports the majority of C11 but some aspects of the language are left out. Not all standard libraries are supported, some types are not recognized, and some functions are not properly handled in CIVL.

5.2 Future Work

Adding support for unsupported constructs will increase the number of programs that the transformer can be applied to. The goal is to support all of the current OpenMP specification. By being able to support all of the OpenMP specification, all OpenMP C programs should be able to be transformed by the OpenMP transformer.

Many of the examples that are transformed by the OpenMP transformer are smaller examples. There are efforts going on to be able to verify the large examples that are on the order of 1000-10000s lines of code instead of around 100 lines of code. Being able to verify larger examples will show that the transformer can be applied to any OpenMP program and will be useful in verifying real world programs.

The tests that have been performed on the OpenMP transformer show that the transformer is able to handle the majority of the OpenMP specification. The transformer can easily be modified to add support for any unsupported or new

functionality. The OpenMP transformer serves as a useful addition to the CIVL framework to help support verification of OpenMP C programs.

Bibliography

[1] T. Mudge, "Power : A First-Class Architectural Design Constraint," *Computer*, vol. 34, no. 4, pp. 52–58, 2001.

[2] Introduction to parallel computing. [Online]. Available: https://computing.llnl.gov/tutorials/parallel_comp/#Overview

[3] Pthreads helloworld. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads/samples/hello.c

[4] Openmphelloworld. [Online]. Available: https://computing.llnl.gov/tutorials/openMP/samples/C/omp_hello.c

[5] OpenMP Architecture Review Board, "OpenMP application program interface version 4.0," 2013. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[6] "Lawrence Livermore National Laboratory OpenMP tutorial," https://computing.llnl.gov/tutorials/openMP/exercise.html, accessed Feb. 8, 2015.

[7] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Can traditional programming bridge the ninja performance gap for parallel computing applications?" *SIGARCH Comput.*

Archit. News, vol. 40, no. 3, pp. 440–451, Jun. 2012. [Online]. Available: http://doi.acm.org/10.1145/2366231.2337210

[8] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[9] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. [Online]. Available: http://www.eecs.berkeley.edu/~ksen/papers/cacm13.pdf

[10] "CIVL: Concurrency Intermediate Verification Language," https://vsl.cis.udel.edu/civl, Accessed Apr. 17, 2015.

[11] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers, "Civl: The concurrency intermediate verification language," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 61:1–61:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807635

[12] S. F. Siegel and T. K. Zirkel, "A Functional Equivalence Verification Suite," http://vsl.cis.udel.edu/fevs, accessed Feb. 6, 2015.

[13] "Lawrence Livermore National Laboratory Message-Passing Interface (MPI) exercise," https://computing.llnl.gov/tutorials/mpi/exercise.html, accessed Feb. 8, 2015.

[14] VirginiaTech: Advanced Research Computing, "CUDA," http://www.arc.vt.edu/resources/software/cuda, accessed Feb. 6, 2015. [Online]. Available: http://www.arc.vt.edu/resources/software/cuda

[15] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2010. [Online]. Available: https://developer.nvidia.com/content/cuda-example-introduction-general-purpose-gpu-programming-0

[16] Purdue University, Information Technology: Research Computing, "Carter – User Guide," https://www.rcac.purdue.edu/compute/carter/guide/#compile_gpu, 2008, accessed Feb. 6, 2015. [Online]. Available: https://www.rcac.purdue.edu/compute/carter/guide/#compile_gpu

[17] C. Hathhorn, M. Becchi, W. L. Harrison, and A. M. Procter, "Formal semantics of heterogeneous CUDA-C: A modular approach with applications," in *Proceedings of the 7th Conference on Systems Software Verification, SSV 2012, Sydney, Australia, 28-30 November 2012*, 2012, pp. 115–124. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.102.11

[18] "CUDA Samples," http://docs.nvidia.com/cuda/cuda-samples/, accessed Apr. 15, 2015.

[19] "Lawrence Livermore National Laboratory Pthreads tutorial," https://computing.llnl.gov/tutorials/pthreads/exercise.html, accessed Feb. 8, 2015.

[20] "SV-COMP 2015: Competition on software verification," http://sv-comp.sosy-lab.org/2015, Accessed Feb. 7, 2015. [Online]. Available: http://sv-comp.sosy-lab.org/2015

[21] Center for Development of Advanced Computing, "Programming on Multi-Core Processors Using MPI - Pthreads," http://cdac.in/index.aspx?id=ev_hpc_hypack_mpi_pthreads_overview, accessed Apr. 17, 2015.

[22] P. Balaji, J. Dinan, T. Hoefler, and R. Thakur, "Advanced MPI programming," Tutorial at SC13: International Conference on High Performance Computing, Networking, Storage, and Analysis, Denver, Colorado, November 2013, accessed Feb. 6, 2015. [Online]. Available: http://www.mcs.anl.gov/~thakur/sc13-mpi-tutorial/

[23] M. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.

Appendix A

Openmp CIVL Library

A.1 Support Types

This appendix contains the support library information that can be found on the CIVL webpage[10].

A.1.1 $omp_gteam

This is the global team object. It represents a team of threads that execute in a parallel region. This is where all the information that is needed to correctly execute a parallel region will be stored. The global barrier and worksharing queue for every this is located here.

```
1 typedef struct OMP_gteam {
2   $scope scope;
3   int nthreads;
4   _Bool init[];
5   $omp_work_record work[][];
6   $omp_gshared shared[];
7   $gbarrier gbarrier;
8 }* $omp_gteam;
```

A.1.2 $omp_team

This is a local object that belongs to a single thread. It references the global team object. It includes the local views of all shared data and a local barrier.

```
1 typedef struct OMP_team {
2   $omp_gteam gteam;
3   $scope scope;
4   int tid;
5   $omp_shared shared[];
6   $barrier barrier;
7 }* $omp_team;
```

A.1.3 $omp_gshared

This is a global shared object which has a reference to a shared variable.

```
1 typedef struct OMP_gshared {
2   _Bool init[];
3   void * original;
4 }* $omp_gshared;
```

A.1.4 $omp_shared

This is a local view of a shared object that belongs to a single thread. There is a reference to the global object, and a local copy and a status of the shared object. The type of the status variable is obtained from the type of the original variable by replacing all leaf nodes in the type tree with 'int'.

```
1 typedef struct OMP_shared {
2   $omp_gshared gshared;
3   int tid;
4   void * local;
5   void * status;
6 }* $omp_shared;
```

A.1.5 $omp_work_record

This is the worksharing information that a thread needs for executing a worksharing region. It contains the kind of the worksharing region, the location of the region, the status of the region and the subdomain.

```
1 typedef struct OMP_work_record {
2   int kind;
3   int location;
4   _Bool arrived;
5   $domain loop_dom;
6   $domain subdomain;
7 }$omp_work_record;
```

A.1.6 $omp_var_status

This is an enumeration type for the status of a shared component. Available enumerators are: EMPTY, FULL, MODIFIED.

A.2 Support Functions

A.2.1 Team Creation and Destruction

A.2.1.1 $omp_gteam $omp_gteam_create($scope scope, int nthreads)

This creates new global team object, allocating object in heap in the specified scope. Number of threads that will be in the team is nthreads.

A.2.1.2 void $omp_gteam_destroy($omp_gteam gteam)

This destroys the global team object. All shared objects associated to the team must have been destroyed before calling this function.

A.2.1.3 $omp_team $omp_team_create($scope scope, $omp_gteam gteam, int tid)

This creates new local team object for a specific thread.

A.2.1.4 void $omp_team_destroy($omp_team team)

This destroys the local team object

A.2.2 Shared Variables

None of those variables that comprise a shared object should ever be accessed directly. All access must happen through $omp_read/write, including the local views, status, and shared view.

A.2.2.1 $omp_gshared $omp_gshared_create($omp_gteam, void *original)

Creates new global shared object, associated to the given global team. A pointer to the shared variable that this object corresponds to is given.

A.2.2.2 void $omp_gshared_destroy($omp_gshared gshared)

Destroys the global shared object, copying the context to the original variable

A.2.2.3 $omp_shared $omp_shared_create($omp_team team, $omp_gshared gshared, void *local, void *status)

Creates a local shared object, returning handle to it. The local copy of the shared object is initialised by copying the values from the original variable referenced to by the gshared object. The status variable is initialized to FULL. The created shared object is appended to the shared queue of the $omp_team object.

A.2.2.4 void $omp_shared_destroy($omp_shared shared)

Destroys the local shared object

A.2.2.5 void $omp_read($omp_shared shared, void *result, void *ref)

Called by a thread to read a shared object. ref is a pointer into the local copy of the shared variable. The result of the read is stored in the memory unit pointed to by result. assumes ref is a pointer to a scalar.

A.2.2.6 void $omp_write($omp_shared shared, void *ref, void *value)

Called by a thread to write to the shared object. ref is a pointer into the local copy of the shared variable. The value to be written is taken from the memory unit pointed to by value.

A.2.2.7 void $omp_apply_assoc($omp_shared shared, $operation op, void *local)

Applies the associative operator specified by op to the local copy and the corresponding shared copy, and writes the result back to the shared copy. This happens in one atomic step.

A.2.2.8 void $omp_flush($omp_shared shared, void *ref)

Performs an OpenMP flush operation on the shared object

A.2.2.9 void $omp_flush_all($omp_team)

Performs an OpenMP flush operation on all shared objects. This is the default in OpenMP if no argument is specified for a flush construct.

A.2.3 Worksharing and Barriers

A.2.3.1 void $omp_barrier($omp_team team)

Performs a barrier only.

A.2.3.2 void $omp_barrier_and_flush($omp_team team)

Combines a barrier and a flush on all shared objects owned by the team.

A.2.3.3 $domain $omp_arrive_loop($omp_team team, int location, $domain loop_dom, $DecompositionStrategy strategy)

Called by a thread when it reaches an omp for loop, this function returns the subset of the loop domain specifying the iterations that this thread will execute. The dimension of the domain returned equals the dimension of the given domain omp_loop_dom.

A.2.3.4 $domain(1) $omp_arrive_sections($omp_team team, int location, int numSections)

Called by a thread when it reaches an omp sections construct, this function returns the subset of the integers 0..numSections-1 specifying the indexes of the sections that this thread will execute. The sections are numbered from 0 in increasing order.

A.2.3.5 int $omp_arrive_single($omp_team team, int location)

Called by a thread when it reaches on omp single construct, returns the thread ID of the thread that will execute the single construct.