University of Glasgow | School of Computing Science

Design and compilation of a C-like front end language for the GPRM

Ross Meikleham

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 26, 2015

Abstract

In the past most software was written with serial computation in mind and increases in processor speeds over time would in turn increase the speed the software ran at. However with the rate of increase in processor speeds declining and manufacturers focusing on adding more processor cores this is no longer the case. The result is that serially written software needs to be rewritten if it wishes to take advantage of multiple processors. This project focuses on designing a language which is familiar to most programmers that describes the composition of serial tasks written in C++ and can be executed in parallel.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

1.1 GPRM

1.1.1 What is the GPRM

The Glasgow Parallel Reduction Machine[1] is a virtual machine framework for multi-core programming using a task-based approach. It allows the programmer to structure their programs as a separation of task-code (written as C++ classes) and communication code.

Communication code is currently written in a language called GPIR (Glasgow Parallel Intermediate Representation). GPIR code controls how tasks communicate with one another and whether groups of tasks can be evaluated sequentially or in parallel. GPIR code is compiled down further to GPRM byte-code which is evaluated by the GPRM virtual machine.

The GPRM uses task nodes which consists of a task kernel and a task manager.

Task code is represented as a task kernel. A task kernel is a self contained unit, typically represented as a C++ class. To create a task kernel, the C++ class needs to be in the *GPRM::Kernel* name-space.

Communication code is represented as a task manager. A task manager "coordinates" communication between one or more task kernels, and is represented as a function which can be called from a C++ program.

When building, the GPRM packages the Task Kernels with GPRM generic code into a library. During this process the compiler analyses all classes and methods in the Task Kernels and maps them to numeric constants. This process allows GPIR code to call Task Kernels. GPIR code is also compiled down to GPRM byte-code packaged into the library. A serial C++ program when compiled is then linked to this library and a call to the GPRM's "run" method executes the GPRM byte-code.

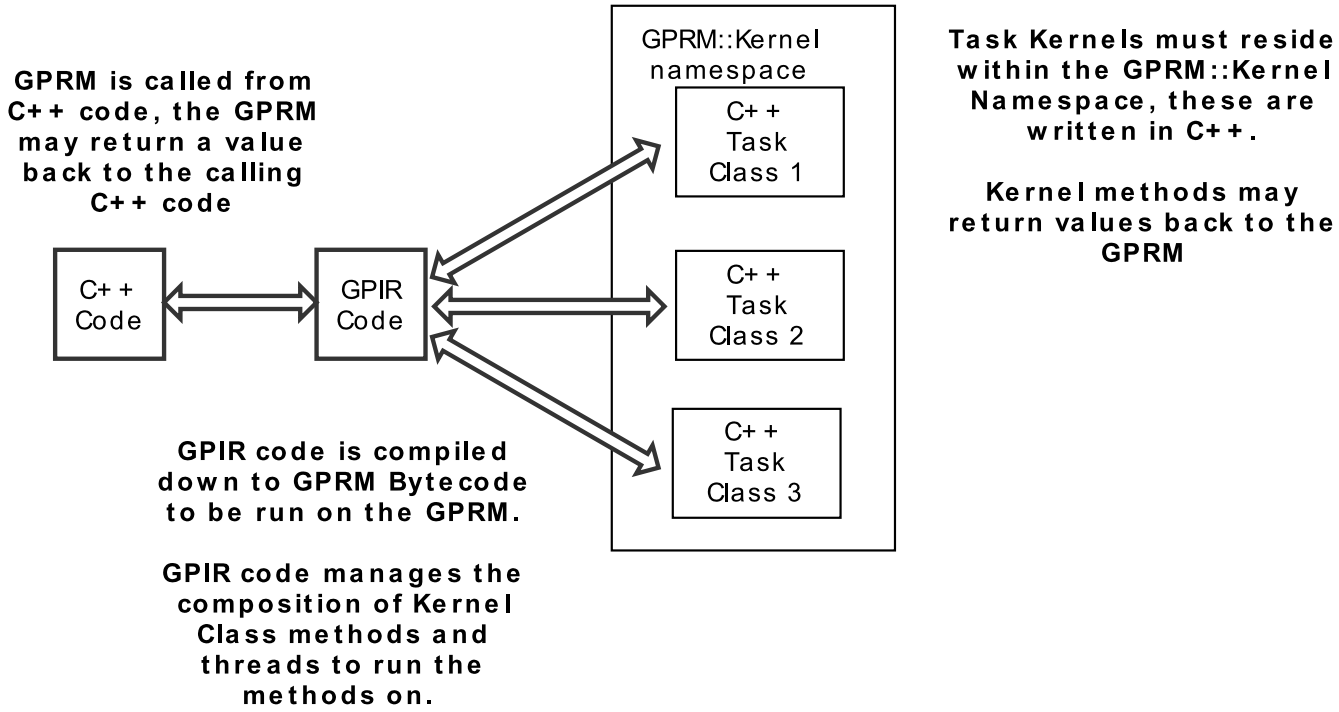


Figure 1.1: A simple overview of the GPRM framework

1.1.2 The GPIR language

The GPIR language is a purely functional S-expression based language that is evaluated in parallel by default with optional sequential evaluation semantics.

Tasks in GPIR are post-fixed with a thread number to indicate to the GPRM runtime which thread the task should run on. For example a simple GPIR program which adds numbers in parallel:

```
1 (begin
2     +[0] (+[0] '3 '2) (+[1] '4 '10)
3 )
```

The two nested additions are performed in parallel, with the first being mapped onto thread 0, and the second being mapped onto thread 1. When they've both been evaluated, then the outer addition will add the results on thread 0.

Quoting

Like in Scheme and Lisp, quoting an expression defers the evaluation of it. This is useful for performing sequential evaluation.

```
1 (seq
2     '(obj.m1[0] '1)
3     '(obj.m2[0] '2)
4 )
```

Due to the parallel evaluation of GPIR, if the expressions in the seq block aren't quoted then they will be evaluated in parallel instead of being deferred to be evaluated by the `seq` function.

Also literal values don't need to be evaluated, they should be deferred and passed to tasks which is the reason the numbers are quoted in these examples.

Registers

The GPRM has registers which can be written to or read from.

```
1 (seq
2     '(reg.write[0] '1 obj.m1[0])
3     '(obj.m2[0] (reg.read[0] '1))
4 )
```

This program writes the result of the `obj.m1` call to register 1, then reads the result from register 1 and passes it into the `obj.m2` call.

The GPIR language has other keywords and features but these are the ones that are important for understanding the examples and design choices made for this project.

1.2 Project Aims

The GPIR language isn't ideally suitable for programming in. For one it requires the programmer to manually manage which thread each task is allocated to. The language is also inconsistent with the C++ language used for the other parts of the framework (Calling code and Class Kernels). A language closer to C/C++ is more consistent with the entire framework.

The aim of this project is to design a C-like language which can be evaluated in parallel by default, and build a compiler for it. The compiler should be able to compile this new language down to GPIR code. The new language will be called " Glasgow Parallel C" or "GPC" for short.

The language should be easy to pick up and write programs in for anyone familiar with the C/C++ languages. To achieve this the language should be as close to C/C++ as possible.

The language should also abstract away the details of allocating tasks to threads, this job should be part of the compiler.

1.3 Current C/C++ Parallel Programming Models

By researching available C/C++ parallel programming frameworks/language extensions we can determine possible features and design choices that may suitable for the GPC language.

1.3.1 Cilk Plus

Cilk Plus is a general purpose programming language based on Cilk++[2]. It extends the C++ language with features such as parallel for loops and spawning functions in parallel using a "fork-join" model to achieve task-parallelism.

One of the main principles of the Cilk language is that abstraction is important and that the programmer should use provided constructs to expose the parallelism in their application. This allows the programmer to be free to focus on what the code is allowed to execute in parallel and not worry about the underlying details of manually managing threads. The run-time should then have the responsibility of scheduling the threads and dividing work between processors.[3].

Cilk Plus introduces 3 new keywords on top of the C++ language[4]:

- *cilk_for* - Parallelizing for loops, uses the exact same syntax as the standard C++ for-loop with some restrictions.
- *cilk_spawn* - Indicate that a given function can run in parallel with the remainder of the calling function.
- *cilk_sync* - Wait for all spawned calls to finish.

Cilk Plus applications have "serial semantics"[4], this means that the results of an application run in parallel with Cilk Plus would be exactly the same if it were run serially (*cilk_spawn* becoming a function call, removing *cilk_sync* statements, and replacing *cilk_for* with ordinary for loops).

Cilk Plus makes use of pragmas to indicate to the compiler that a for loop contains data parallelism [3].

Cilk Plus also introduces a new operator **[:]** to select array sections[5]. This operator allows for "high level" operations to be performed on arrays, and can help the compiler vectorize parts of code.

The Cilk Plus runtime makes use of "task stealing" for dynamic load balancing[3]. This means that if one thread is idle the scheduler can reassign work assigned to be completed by a busier thread. The outcome of this is that the programmer doesn't have to worry about the specifics of which threads to map tasks to, and it can be left to the runtime itself.

1.3.2 Open MP

OpenMP (Open Multi-Processing) is a language extension available for C, C++ and Fortran which allows for shared memory multi-processing. This is achieved by the use of compiler directives (more specifically in C/C++ this is done through the preprocessor using pragmas) and the OpenMP API[6].

OpenMP's use of pragmas for parallel programming means that parallel code keeps sequential semantics and any compiler which doesn't implement OpenMP extensions can still compile the code by ignoring the pragmas. The results of executing the program serially without the pragmas should be exactly the same as when the sections are parallelized. Another benefit is that sections of serial programs can be parallelized by adding pragmas and existing code doesn't need to be modified.

For example, given a simple for loop below:

```
1 for(int i = 0; i < 10; i++) {
2     arr[i] =  do_calculation(i);
3 }
```

The above code can be parallelized by simply adding a pragma above the for loop:

```
1 #pragma omp parallel for
2 for(int i = 0; i < 10; i++) {
3     arr[i] =  do_calculation(i);
4 }
```

OpenMP also supports task parallelism as of version 3.0[7]. The specification for OpenMP does not specify how the scheduler should work, and no specific implementations of OpenMP appear to have implemented a task stealing scheduler.

A downside to using pragmas is that accidentally missing certain directive keywords may cause undesirable program behaviour such as unnecessary parallelization[8], and no warnings will be given by the compiler.

1.3.3 Intel TBB

Intell TBB (Intell Thread Building Blocks) is a portable C++ template library for task parallelism. It contains a range of concurrent algorithms, containers, and it's own task scheduler to achieve this[9].

Operations are treated as tasks, and the task scheduler has the job of dynamically allocating these tasks to individual cores which abstracts the specific details of allocating threads from the Programmer. Like Cilk Plus, Intell TBB's task scheduler also implements task stealing for dynamic load balancing[10].

Intell TBB relies on generic programming which allows for writing parallel algorithms based on requirements on types. Parallel code can be written once and can work with lots of different types without having to rewrite the algorithm for each type [11].

1.3.4 Wool

Wool[12] is a library written for C that supports task parallel programming. It is inspired by Cilk and introduces three forms of synchronization (Create, Exit, and Join). Instead of being a language extension like Cilk, macros and inline functions are used for low overhead in task operations.

Wool relies on tasks having low overheads, and focuses on having low overhead of task creation for fine grained task parallelism. Tasks can only synchronize when a task is created or completes execution. To reduce overhead, if every thread in the system is busy, further tasks are executed sequentially as procedure calls which saves overhead in having to create actual threads.

Wool implements a work-stealing scheduler. Each thread is given its own task dequeue. A `SPAWN` operation pushes a task onto one of the queues, and a `SYNC` operation removes a task from the tail of the queue. Tasks are taken from the front of the queue and executed. Only the owner thread of the queue can remove a task from that queue. If a thread's task dequeue is empty it can steal a task from the front of one of the other queues.

Chapter 2

Design

The goal is to create a C-like language which then compiles down to GPIR code. The language should be as C-like as possible, ideally it should be as much of a complete subset of C++ as possible.

C++ is a statically typed, imperative language which is sequentially evaluated by default. GPIR is a dynamically typed functional language which is evaluated in parallel by default. These two languages use two completely different paradigms, so the language design has to ensure that GPC is like C++ and can be compiled to GPIR without too much trouble.

2.1 GPC Language Design Decisions

2.1.1 Parallel Evaluation

GPIR is parallel evaluated by default, with a `seq` function which evaluates the given arguments in sequential order. Mapping GPC code into GPIR code which can be evaluated in parallel should be possible, and allows GPC to take advantage of parallel evaluation by default. GPC should also have a method of sequentially evaluation of multiple statements.

2.1.2 Serial Semantics

Having serial semantics is helpful to the programmer as it makes it easy to reason about the behaviour of parallel code. This is due to the ability to run the entire parallel code in a single thread as if it were sequential code.

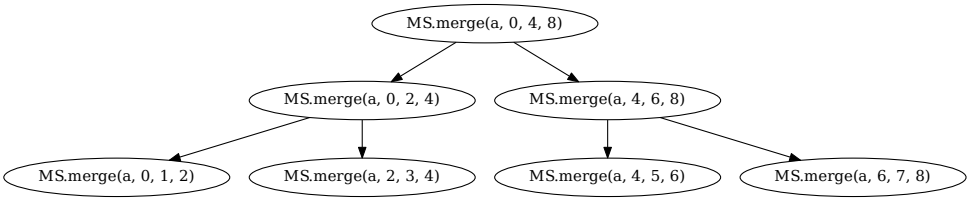
2.1.3 Purely Functional

In the GPRM jumping to labels is an expensive operation, to avoid this function calls will need to be inlined, and loops unrolled as much as possible. The execution path will also need to be known during compile time to achieve this.

Another one of the major benefits of being able to compute the execution path at runtime is that recursive functions can be more efficiently parallelized by the compiler. The following merge sort example is used to illustrate this.

```
1
2  //Use Sort class from Kernel
3  GPRM::Kernel::Sort MS;
4
5  int size = 8
6
7  void merge_sort(int *a, int low, int high) {
8      if (low + 1 < high) {
9          mid = (low + high) / 2;
10
11         seq {
12             par {
13                 merge_sort(a, low, mid);
14                 merge_sort(a, mid, high);
15             }
16             MS.merge(a, low, mid, high);
17         }
18
19 )
20
21 void GPC_merge_sort(int *a) {
22     merge_sort(a, 0, size);
23 }
```

The merge kernel calls be be reduced down to a tree:



From the bottom up, the merge operations in the row can be performed in parallel, knowing the execution path at compile time allows for the compiler to map these to separate threads.

However if the language is Turing-complete deciding the execution path of a given program is essentially the Halting Problem. It has been proven that the solution to this problem is undecidable[13].

To avoid this problem, the language needs to be restricted to a point where it can still be useful and the execution path decided at compile time.

If the language is purely functional with no side effects then this avoids the problem, however a program with no side effects is useless. There is one major feature that must be in the language that invokes side effects which is method calls for kernel objects.

To achieve this the type system can enforce that anything returned from a kernel object method call's type then has to be in the `GPRM::Kernel` namespace, and anything passed into a kernel object method call is "cast"into the `GPRM::Kernel` namespace. This allows passing impure values between kernel methods but restricts them being used for conditional statements. Essentially we still have side effects but they're restricted to only being in Kernel space and not in the GPRM. The execution of GPC code is then purely functional and the execution path of code can be determined at compile time without too much difficulty.

2.2 GPC Language Features

This section explains the features of the language with respect to the design decisions in section 2.1.

2.2.1 Syntax

The syntax aims to be as close to C/C++ as possible. Statements must end with a semicolon. All variables must be statically typed. Blocks are surrounded in braces. Case sensitivity will be enforced.

2.2.2 New Operators

Two new operators not currently present in C++ `seq` and `par` are introduced. These are placed before a block of statements to determine whether each individual statement within the block are to be evaluated in sequential or parallel. By default a block of statements are evaluated in parallel, but the `par` keyword makes it more explicit for readability in the case of lots of nested `seq`/`par` blocks.

2.2.3 Objects

Objects can be declared in the top level scope, and must be in the GPRM::Kernel namespace. For example declaring a test object from a class called Test in the `GPRM Kernel` namespace:

```
1 GPRM::Kernel::Test test;
```

Standard C++ method calling syntax can be applied to objects.

```
1 test.m1();
```

2.2.4 Compatibility With C++

Making the language completely compatible with C++ in that all GPC code can be compiled with a C++11 compatible compiler allows programmers that are familiar to C++ write GPC code with little difficulty. As long as they are aware of the restrictions that GPC has compared to C++. However the new keywords `seq` and `par` are not in the C++11 language.

Currently preprocessor directives aren't supported in GPC so the compiler can just ignore lines starting with `#`, and define `seq` and `par` as no-ops. For example this snippet of code compiles with both the GPC compiler and should compile with all C++11 compilers:

```
1  #include "GPRM/Kernel/Test.h"
2  #define seq
3  #define par
4
5  GPRM::Kernel::Test test;
6
7  int entry_fn() {
8      seq {
9          int a = test.m1();
10         int b = test.m2(a);
11         par {
12             test.m3(b);
13             test.m3(b + 1);
14         }
```

```
15          return 0;
16      }
17  }
```

When this code is compiled with a C++ compiler it will remove the `seq` and `par` keywords and plain blocks will be left. Essentially this should generate a serial version of the program which should generate the exact same results as the version compiled by GPC and run on the GPRM. This supports the design decision of serial semantics.

This is useful for implementing the compiler as it allows for testing that the code generated by GPC is generating the correct results by comparing it to the C++ version. It also allows for easier porting of programs already written in C++ to GPC. Programmers can use tools already made for C++ to debug their serial code before attempting to run it in parallel on the GPRM.

However this method restricts the possible implementation of pragmas into the GPC language in the future. A possible improvement would be to change from the Cilk Plus approach of adding extra keywords to the language to an openMP approach of having `seq` and `par` pragmas instead. Then the language would be fully compatible with C++ without needing to modify the GPC code.

2.2.5 The Type System and Kernel restrictions

C++ types such as string, char, bool, int, and double are included. Pointers, and "multilevel" pointer types are included (e.g. int**, char*). However pointers are restricted in that the address of a variable cannot be taken. Adding and subtracting integers from pointers is allowed. Usually pointers are passed into the GPRM from the C++ caller to represent an Array.

Return values from kernel method calls are implicitly placed in the `GPRM::Kernel` namespace.

For example:

```
1  int x = obj.method(5);
```

This is implicitly cast to:

```
1  GPRM::Kernel::int x = obj.method(5);
```

If a binary expression involved a Kernel value and a "pure" value then the "pure" value is implicitly upcast before the operation takes place. For example:

```
1  bool y = obj.m1(10) == 5;
```

is implicitly cast to:

```
1  GPRM::Kernel::bool y = ob.m1(10) == 5;
```

This feature stops impurity in the GPC code execution, as the type system should be able to stop Kernel types being used in conditional statements.

For example this is not allowed:

```
1
2  seq {
3      int x = obj.method(5);
4      if (x == 10) {
5          //Do Stuff
6      }
7  }
```

If statements must take a "pure" boolean type as its conditional. `x == 10` is a `GPRM::Kernel::bool` type, so this raises a type error.

2.2.6 Operations

Most basic binary arithmetic operations are included i.e. (+, -, *, /, %, ==, !=, &&, || , <<, >>, &, |, ^) as well as unary operations (−, ~, !).

(+=, ++, –, -=) are not included, due to the single assignment rule. Although an exception is made for the "afterthought" of the for loop construct, in which the integer loop variable can be incremented with += or decremented by -=.

2.2.7 Functions

Basic support for defining and calling functions is supported, and function syntax is exactly the same as C. However there is currently no support for function pointers or C++11 lambdas.

2.2.8 Single Assignment

Variables in GPC can only be assigned once per scope to keep the language functional.

for example the following isn't allowed:

```
1 int i = 0;
2 int i = i + 1;
```

Since `i` is already in scope, it cannot be redefined in the same scope.

The following code is allowed:

```
1 int i = 0;
2 {
3    int i = i;
4 }
```

Since `i` inside the block is declared in a new scope.

Also variables must be assigned when they are declared, for example the following is not allowed:

```
1 int x;
2 x = 5;
```

2.2.9 For-Loops

For loops have the same syntax as C/C++ for loops with some restrictions:

- There is a single loop variable which must be declared inside the loop, and must be an integer.
- The conditional expression must result in a "pure" boolean type, a *GPRM::Kernel::bool* type is not allowed.

- The "afterthought" of the for loop must consist of the loop variable with either the += operator or −= operator with a "pure" integer value on the right hand side. This is the only place these binary operations are allowed to occur in a GPC program.

These restrictions and properties are in place to keep the functional "purity" of GPC code, allowing complete compatibility with the C++ language, and keeps the property of GPC programs having serial semantics. At compile time the loop is fully unrolled, these restrictions also allow for detection at compile time whether or not the loop is infinite.

It's worth noting that these restrictions are similar to the restrictions on the *cilk_for* loop. The *cilk_for* loop must declare a single initial loop variable, the conditional expression must compare the loop variable with a constant "termination expression", and the "afterthought" must either increment or decrement the loop variable by some amount [14].

An example of a for loop is as follows:

```
for(int i = 0; i < 5; i+=1) {
    obj.m1(i);
}
```

During compilation this loop is fully unrolled and is equivalent to the following:

```
par {
    obj.m1(0);
    obj.m1(1);
    obj.m1(2);
    obj.m1(3);
    obj.m1(4);
}
```

2.2.10 Top Level Statement Restrictions

Top Level statements are restricted to being either object declarations, function definitions, or constant variable assignments. This is partly to be like C++ and also to remove any ambiguity on how top level statements are evaluated.

2.2.11 Entry Function

Since the GPIR function is called by C++ code, it's not preferable to name the function "main". Also the C++ code may be calling more than one GPIR function during its lifetime so a static name is also not preferable. The GPIR code entry function has the same name as the GPC source file it is in. For example the entry function for "test.gpc" would be called "test". This method of determining an entry function is not ideal, but is easy to change in the future if needed.

Chapter 3

Implementation

3.1 Implementation Language Choice

The compiler is implemented in the Haskell programming language.

During complication the need to traverse trees usually occurs quite often. Functional languages like Haskell are suited to this task due to features such as pattern matching and tail-end recursive optimization which make traversing trees efficient and simple to implement.

The Glasgow Haskell Compiler is available for most platforms, most importantly for Windows, OSX and Linux on x86 architectures. This allows the compiler to be portable across these platforms provided the libraries used to build the compiler are portable.

Haskell has support for algebraic data types, these makes ASTs(Abstract Syntax Trees) simpler to implement. For example the AST for a very simple expression can be represented as follows:

```
1
2       data Expression =
3             Add Expression Expression
4           | Negate Expression
5           | Const Integer
```

The equivalent in an Imperative/OOP language (in this case Java) would be the following:

```
1
2   abstract Class Expression {}
3
4   class Add extends Expression {
5       public Expression left;
6       public Expression right;
7       public Add(Expression l, Expression r) {
8           left = l;
9           right = r;
10      }
11  }
12
13  class Negate extends Expression {
14      public Expression expr;
15      public Negate(Expression e) {
16          expr = e;
17      }
18  }
19
20  class Const extends Expression {
21      public int value;
22      public Const(int v) {
```

```
23          value = v;
24      }
25  }
```

The Haskell version is much clearer on the structure of the tree, and takes much less code to implement.

It is also to easily extend Haskell data types to include custom annotations. For example storing the source code position for parts of expressions which is useful for reporting errors to the user.

Due to the pure function nature of Haskell, parts of compilation can easily be performed in parallel. For example during type checking each GPC function can be type checked in parallel and this can even be subdivided further into blocks within functions. For this project speed of compilation is not a major concern, but in the future if compilation ever needs to be faster then this option is always available.

Haskell also has powerful libraries for parsing source code such as Parsec which is a parser combinator library. Parsec allows combinator parsers to be written in the Haskell language itself avoiding the complexity of integration of different tools and languages[15].

3.2 Tools and Testing

3.2.1 Cabal

Cabal (Common Architecture for Building Applications and Libraries) is a system for building and packaging Haskell libraries and programs[16]. This system can manage the project library dependencies, and automatically download and install missing dependencies. It can also build and install the compiler on the system, and run unit tests.

3.2.2 Testing

For Unit testing the HUnit[17] library will be used, this can be integrated with Cabal to easily run all the required unit tests.

One form of testing used for this project is testing each individual component (e.g. The Parser or the Type Checker). Each component in the compiler can easily be uncoupled from one another due to the linear nature of compilation.

Another form of testing is writing GPC source files which should compile, and GPC files which should fail compilation at a certain stage. The compiler is then invoked during testing on all of these files to check whether all the source files which should compile do in fact compile with no errors, and all the source files which should raise an error do not compile.

An upside to this method is that testing this way is flexible in that tests aren't coupled with the implementation internals of the compiler. The only way that these test would need to be changed was if the design of the language itself would need to be changed.

A downside to this method is that without manually checking the errors raised by the tests that should fail; the source may generate an error which is unrelated to the error that is being tested. This is why some testing of each internal component is done alongside this method.

3.2.3 Code Coverage

The Haskell HPC[18] (Haskell Program Coverage) library allows for recording code coverage over different modules during testing. This can be integrated with cabal unit testing to automatically generate these results. The usefulness of this allows for checking which sections of code still need to be tested and assists in writing further unit tests.

3.3 Compiler Structure

Compilation is split up into multiple stages or "passes", it is possible to compile in one pass but separating each specific section of compilation allows modularity and decoupling.

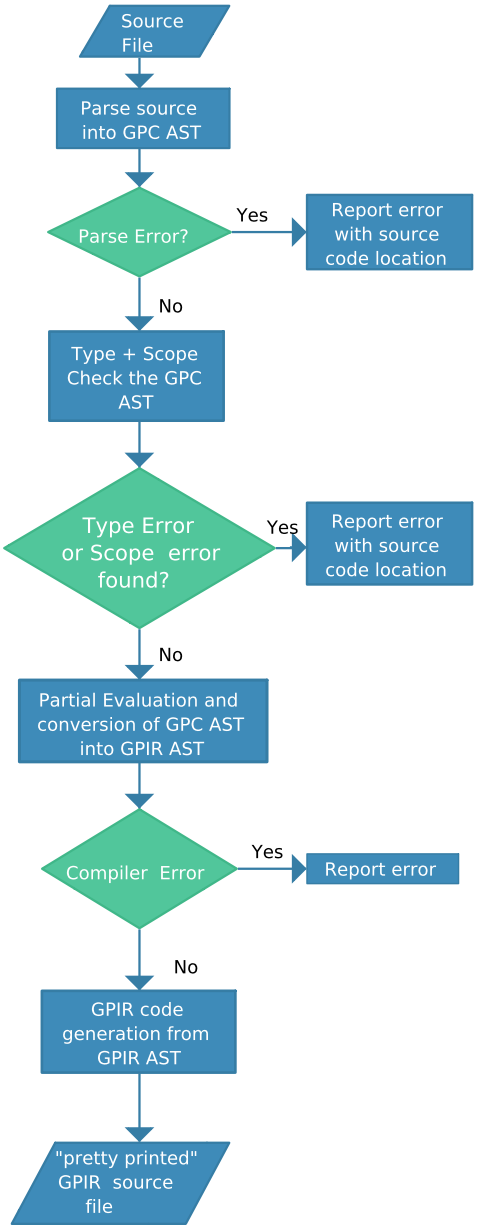


Figure 3.1: Flowchart illustrating the stages of the GPC compiler.

3.4 Parser/Lexer

The Parsec library combines Parsing and Lexing into one stage.

Given the GPC source file the intention is to parse the file into an AST to hopefully eventually be transformed into GPIR source code. It's also useful to store the original source position to provide error information during a further compilation stage, to achieve this the source position info is read from Parsec into an Annotated AST as the tree is being built up.

Parsec does most of the work during this stage, including providing error messages for "expected" values to be found in source positions and source position information. Most of the work implementing this stage is building the parser combinator functions and composing them together to be able to build the AST.

3.5 Type and Scope checking

During type checking the types of identifiers from the current scope being used in expressions need to be known. Since the scope needs to be kept track of while type checking it makes reasonable sense to check for scope errors in the same stage.

The goal of this stage is to ensure that the static typing of the source file is enforced (e.g. attempting to assign a bool value to a variable of type int should not happen) and, prevent "logic" errors at compile time (e.g. adding 2 bool values together). Scope checking is also important as identifiers being used within the program need to be binded to an expression of some sort, and the "single assignment" rule in the GPC language needs to be enforced.

There are two separate "types" of scope in a GPC program. The first is the top level scope and the other is at function level scope. Any scope "further" down from function level scope is itself a function level scope. During this stage the top level scope is type and scope checked. Afterwards each individual function is type and scope checked.

For checking over top level statements the following Haskell record is used to keep track of identifier types, objects, and functions encountered:

```
1 type VarTable = M.Map (Ident SrcPos) (Type SrcPos)
2 type FunTable = M.Map (Ident SrcPos) (Type SrcPos, [Type SrcPos])
3 type ObjectTable = M.Map (Ident SrcPos) (Objects SrcPos)
4
5 data MainBlock = MainBlock {
6     _tlFuncDefs      :: FunTable, -- ^ Function Definitions
7     _tlVarTypes      :: VarTable,  -- ^ Top Level Constant variable types
8     _objects         :: ObjectTable  -- ^ Table of current Kernel objects declared
9 } deriving (Show)
```

The type checker needs to check that the are no instances of duplicate functions or duplicate top level variables. Once this is done, and details of functions and top level variables have been stored, each individual function can be type checked.

A slightly different structure is needed to type check functions.

At any point in time the type checker needs to know the following:

- Variables that are currently in scope, and their types
- Functions that are available to call, their argument types, and return types

- Objects that are available to call methods on
- The current function the type checker is in

These values are stored using the following Haskell record:

```
1
2  data CodeBlock = CodeBlock {
3      _currentFun :: Ident SrcPos, -- ^ Name of Function block is in
4      _funcDefs   :: FunTable, -- ^ Function names and return/argument types
5      _prevVars   :: VarTable, -- ^ Identifiers visible in current scope with types
6      _curVars    :: VarTable  -- ^ Identifiers declared in current scope
7  } deriving (Show)
```

When a function is being type checked a new CodeBlock instance is created from a MainBlock instance as some of the top level information is needed. The details of the function that is being entered is stored in `_currentFun`, details of all available functions are stored in `_funcDefs`, and all top level variables are stored in `_prevVars`. `_curVars` is left as an empty map as the type checking of the function hasn't begun yet. Top level objects are also stored in `_curVars` as an "object" variable type.

Whenever a new scope is encountered by either calling a function, or entering a seq/par block; a new CodeBlock structure is created using the current structure. The new `_curVars` is set to the empty map as no variables have been encountered yet, the `_funcDefs` are copied as all functions are on the top level so they don't change. The `_currentFun` is copied if entering a block, otherwise if entering a function the function name and source position are copied.

The value of the new `_prevVars` is a little more complicated to work out. Any key-value pairs in the current `_curVars` are stored plus any key-value pairs in the current `_prevVars` in which the key isn't present in the current set of `_curVars` keys. This is because the identifiers in `_curVars` scope are visible over the identifiers with the same name in `_prevScope`. Haskell's union operation on maps discards the key-value pairs in the second map for keys that are present in the first map, so this is trivial to implement.

When type checking a function every statement in the function is type checked. For every statement every expression within the statement is type checked, this is implemented by traversing the Statement AST and checking the scopes of identifiers as well as checking expected types match the actual types of each expression.

Objects which are declared are not checked to see if they actually exist. Neither are method calls which means the argument types and return types when calling them cannot be determined at compile time. This is due to the fact Objects are instantiated from C++ classes. The C++ class of the object would need to be checked for methods available as well as argument and return types. This is already implemented in the GPRM, so an error will occur further down to compile "chain" or during runtime.

If a type or scope error is encountered, an error message determining the type of error, and the source position of the error is returned. Otherwise an empty tuple is returned. When type and scope checking the original AST doesn't need to be modified, only verified that it follows the type and scope rules of the language.

3.6 Interpreting

The goal of this stage is to run through the execution path the GPIR code will take from the entry function, and partially evaluate the code as much as possible while generating the GPIR AST. This stage involves **Sparse Conditional Constant propagation** (**SCC**) like optimizations.

SCC[19] is an optimization algorithm applied in compilers. It involves removing dead code and performing **Constant Propogation** which involves replacing identifiers which can be evaluated to constant values at compile

time with those values. As well as evaluating the results of constant expressions and further propagating them. Usually this involves interpreting the AST. How this stage differs from purely performing **SCC** optimizations is that a new AST is also being generated as expressions are being evaluated. Also branches are always able to be evaluated at compile time. Due to these reasons this section of the build process is called "interpreting". However **SCC** can only be applied to an immediate representation (in this case an AST) which is in **Static Single Assignment form** (**SSA**).

SSA[19, 20] requires that every variable is assigned exactly once and is defined before being used. The GPC language already enforces single assignment, and that variables are assigned when they are defined (see section 2.2.8). Scope checking in section 3.5 proves that all variables used in the GPC AST passed to the interpreter have been defined. Therefore the GPC AST given to the interpreter is already in **SSA** form.

Just before interpreting the GPC AST is transformed slightly into a similar AST with a couple of differences. One is that annotations are not present (since source position information is not needed anymore), and type information is stripped (since the program has been proven to not have any type errors), also objects in the top level scope are stripped (since scope checking proved that all methods are called on objects that exist, and whenever a method is called, the name of the object is part of the call).

Initially the values of all top level assignment statements need to be stored as constants before executing, as well as the details for every function.

While interpreting a state is needed to determine actions taken during certain sections, as well as to evaluate expressions. The following Haskell record is used:

```
1
2  type ConstVarTable = M.Map Ident Literal
3  type FunTable = M.Map Ident ([Ident], BlockStmt)
4  type VarRegTable = M.Map Ident Integer
5
6  -- ^ Current State of the Block we are currently in
7  data CodeGen = CodeGen {
8      _funTable    :: FunTable,  -- ^ Store symbol tree for functions
9      _constTable  :: ConstVarTable, -- ^ Store constants in scope
10     _varId       :: Integer, -- ^ Current variable id for mapping to registers
11     _varRegTable :: VarRegTable, -- ^ maps variable identifier
12     _threadCount :: Integer, -- ^ Current thread number to map
13     _maxThreads  :: Integer,  -- ^ Maximum number of threads
14     _seqBlock    :: Bool, -- ^ Whether or not current block is sequential
15     _isReturning :: Bool -- ^ Whether the state of the current block is in a return
16 }
```

3.6.1 Registers

Values which can be fully evaluated at compile time can be substituted into the GPIR code with their literal value when they are used.

For example:

GPC Code

```
1 seq {
2     int x = 4 * 5;
3     obj.m1();
4     obj.m2(x + 3);
5 }
```

Compiled down to GPIR code

```
seq (
     '(obj.m1[0])
     '(obj.m2[0] '23)
)
```

The value of `x` is able to be calculated to be 20, so whenever `x` is used in the scope it can simply be replaced with the literal `20`.

However, some variables will not be able to be fully evaluated at compile time. So results will have to be written to GPIR registers.

For example:

GPC Code

```
seq {
    int x = obj.m1();
    obj.m2(x);
}
```

Compiled down to GPIR Code

```
seq (
     '(register.write[0] '1 'obj.m1[0])
     '(obj.m2[0] (register.read[0] '1))
)
```

The value of `x` can't be known at compile time so it is written into register `1`, and then read from the same register when it is needed.

During interpreting, there needs to be no conflict between registers (e.g. if a value is stored in register 1 that will need to be used later, register 1 cannot be written to until that value is out of scope). There's no hard limit on registers so a simple method of just incrementing the register count every time a value needs to be stored is implemented, although this may possibly cause a lot of unnecessary memory usage.

`_varRegTable` is used to store the mappings of variable names to register numbers. Conflicts between variables with the same name in different scopes is not a problem, as the register table in the inner block is thrown away once the scope is left.

3.6.2 Sequential And Parallel Block Differences

A sequential block translates to something different in GPIR than a parallel block.

For example, these two snippets compile to different GPIR code despite the blocks containing the same code:

GPC Parallel

```
par {
    obj.m1();
    obj.m2();
}
```

GPC Sequential

```
seq {
    obj.m1();
    obj.m2();
}
```

GPIR Parallel

```
(par obj.m1[0] obj.m2[1])
```

GPIR Sequential

```
(seq `obj.m1[0] `obj.m2[1])
```

Sequential statements need to be quoted, which is why the record contains the `_seqBlock` flag, so the interpreter can generate the correct GPIR code.

3.6.3 Thread Mapping

Each task in the GPRM must be mapped to a thread. There is a function in the Interpreter called `getThread` which calculates what the next thread number should be to map the task to based on the current block state.

Currently this function uses a simple incremental scheme and rolls around modulo style after the max number of threads have been specified. The compiler attempts to work out the maximum number of cores on the machine if a thread number isn't given, and uses this number to determine the max number of threads.

If a different scheme is needed the contents of the function will need to be changed, a possible future improvement may be to pass a function to the interpreter when passing it the AST to determine the counting scheme. This would allow for multiple different schemes to be chosen and possibly tuned depending on the type of application.

3.6.4 Branching

When encountering an if statement the interpreter must be able to evaluate the condition and generate a true or false value. The interpreter will only evaluate the statement within the if statement if the condition is true. In the case of an if-else statement either the first statement will be evaluated or the else statement depending on the value of the condition.

3.6.5 Returning From Functions

Performing a return is a bit tricky.

Some pre-evaluation is needed on blocks of statements when entering a new block. if a return statement is in the current block of statements (not counting sub-blocks) then any statements after that are "removed" and when a return statement is found, it is evaluated as the return expression. What this means is that once the return is met the interpreter "returns" back to the code which executed the previous scope.

There are also two different scenarios when performing a return:

1. **The interpreter is at the top level scope of a function**. Returning is simple in this case, all that is needed to be done is going to the previous scope.
2. **The interpreter is nested within one or more blocks in a function**. Returning in this case involves going up multiple times until the interpreter is out of the scope of the current function.

To deal with these situations a "boilerplate" function is used. This function is called whenever an inner-scope needs to be interpreted. It takes a boolean value which determines if the current block scope is at the "top" of a function, gets the current state and then interprets the block. Once the interpreter returns from interpreting the inner block, the "_isReturning" flag is checked on the returned state. If it is set, and the scope of the current block is at the top of a function, then the "_isReturning" flag is then set for the current block.

When the "_isReturning" flag is set for a block, the evaluator will not evaluate any further statements in the current block. This allows for propagation of a "return" up the block states until the current function is exited.

3.6.6 Interpreting Functions

When a function is called, each argument identifier in scope is replaced by the the expression which was supplied to the function. Then each statement in the function is evaluated.

```
1
2  void fun(int a) {
3      obj.m1(a);
4      seq {
5          int a = 10;
6          obj.m3(a);
7      }
8  }
9
10 fun(5 * 4);
```

In this example when evaluating the functional call with the argument `(5 * 4)`, every `a` which is the same instance as the one in the function arguments is replaced by the expression `(5 * 4)`. The nested `a` is not the same instance and is not substituted.

3.6.7 For Loop Evaluation

For evaluating and unrolling for loops each loop statement needs to be evaluated until the loop variable doesn't satisfy the condition.

- The iterate function in Haskell can be used to infinitely apply the afterthought function of the loop to generate loop variable values for each iteration.
- Then values are taken from this list and are applied to the loops condition until one fails. This creates a list of loop variables values for all iterations.
- This value list is then mapped to the statements in the loop body replacing any instance of the loop variable. This unrolls the loop.
- Then each statement is then evaluated.

It is also possible to detect cases of infinite loops based on the loop variable, the conditional, and afterthought statements, before attempting to unroll the loop.

For example the following for loop will loop infinitely (ignoring overflow of integers, which is undefined in most C/C++ standards).

```
1  for (int i = 0; i < -1; i++) {
2      obj.m1(i);
3  }
```

3.6.8 Partial Expression Evaluation

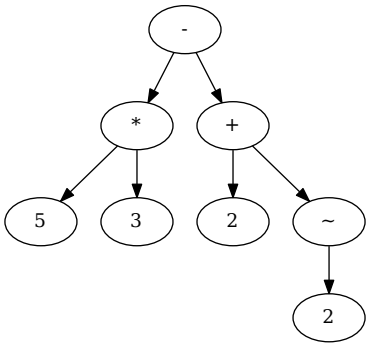
There are two parts needed to evaluate an expression:

1. **Constant Propogation**[19] - All identifiers in the expression which have a value in the constant table are replaced with the respective constant value.
2. **Constant Folding**[19] - Attempt to evaluate as much of the expression with replaced constants as possible

Replacement is trivial, using the constant table for the current scope the expression AST is recursively travelled and any instances of identifiers in the constant table are replaced.

Reduction is a bit more complex, it involves attempting to apply the binary and unary operations to their respective expressions. Expressions can be represented in a binary expression tree. Binary operators are contained in the inner nodes with 2 children, unary operators contained in the inner nodes with 1 child. Leaf nodes contain literal values and variables.

For example the expression `5 * 3 - 2 + ~2` when read in by the compiler is represented by the following tree:



The tree is evaluated through post-order traversal. Once two leaf nodes are found the binary operation in the parent node is attempted to be applied to the two leaf nodes, or if one leaf node is found with no other node the unary operation in the parent node is applied to the leaf.

In the case of a binary operation if both leaf nodes are values which can be calculated at compile time the expression is evaluated, the two leaf nodes are removed and the parent node is replaced with the calculated value. If one or more leaf nodes are values which can only be known at runtime then the expression is not evaluated and the tree doesn't change.

In the case of a unary operation if the leaf node is a value which can be calculated at compile time the expression is evaluated, otherwise the expression is not evaluated and the tree doesn't change.

However this method isn't optimal when some sub-expressions cannot be evaluated at compile time. In this case the expression can be fully evaluated to a single value, but depending on the expression this method can at times reduce the expression down to an expression which can still be further reduced. This problem and a possible solution is discussed in Chapter 5.

3.7 GPIR Code Generation

Once the interpreter is completed it should produce the GPIR AST, from this AST the goal is to output GPIR source code. Since the GPIR language is very simple this task is not too difficult., It involves recursively travelling the tree, ensuring lists containing task operations are surrounded by parentheses, printing quotes when needed, etc.

For this task a "pretty printer" library is used which allows for formatting the output much easier than manipulating strings. The generated GPIR source code is saved into a file with the same prefix as the source but with a ".td" (Task Description) extension.

Chapter 4

Conclusion

4.1 Benchmarks

A merge sort algorithm has been implemented in GPC for 80 million integers with a cutoff value of 2048. The actual sorting and merging is performed by the `MergeSort` kernel class methods `MergeSort.serial_ms` and `MergeSort.merge_two`. The GPC code structures how the Kernel tasks should interact and calls them in parallel.

```
1  GPRM::Kernel::Mergesort MS;
2
3  int overall_size = 80000000; //80 million integers
4  int cutoff = 2048;
5
6  void Sort(int size, int c, int index) {
7      int new_size = size/2;
8
9      if (size <= cutoff) {
10         MS.serial_ms(c%index, size);
11     } else {
12         MS.merge_two(c%index, size, Sort(new_size, 2*c, 2*index), Sort(new_size, 2*c + 1, 2*index));
13     }
14 }
15
16 // Entry Function
17 void GPRM_MergeSort() {
18     seq {
19         MS.array(overall_size);
20         Sort(overall_size, 1, 1);
21     }
22 }
```

This code is compiled into GPIR code by the compiler from the command line by entering `gpcc GPRM_MergeSort.gpc --threads=240` where the `threads` argument is an optional to manually adjust the number of total threads that are available to map tasks onto.

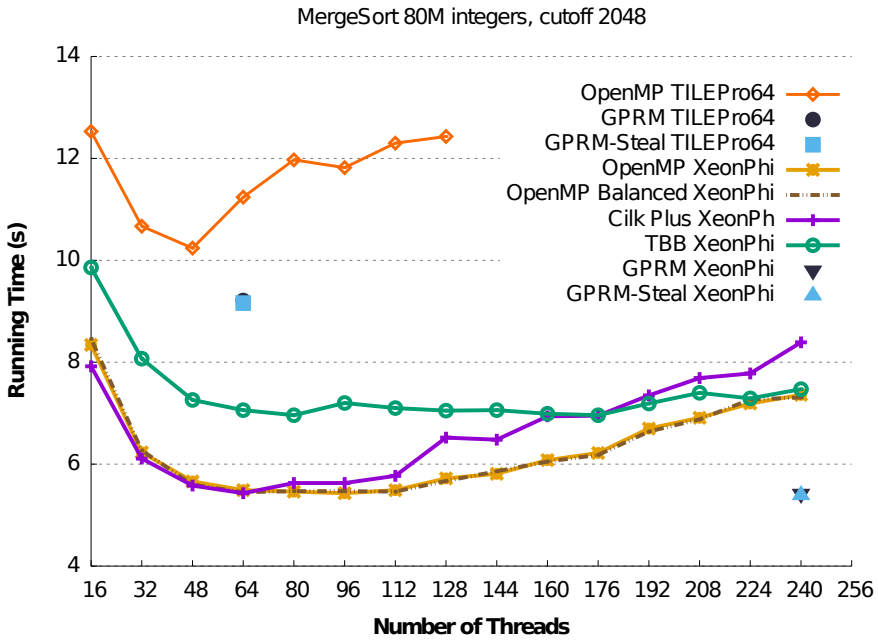


Figure 4.1: Merge sort benchmarks for parallel frameworks [21]

The compiler compiles the merge sort GPC code into GPIR code identical to the GPIR code used to generate figure 4.1. The `MergeSort` C++ Kernel class is also the same one used. From the figure it can easily be seen that the GPRM far outperforms Cilk, TBB, and OpenMP on the XeonPhi, and OpenMp on the TILEPro64 for 240 threads in this specific example.

Another thing to note is that the GPC code is a lot "nicer" than the code for the Cilk, TBB, and OpenMP implementations, as it is almost completely pure C++. It doesn't require the learning of how to use pragmas, or fork-join mechanisms.

4.2 Summary

The main goal of the project was to create a C-like language for the GPRM. I believe that goal has been met as the language designed and implemented in this project is a purely functional parallel evaluation language which is an exact subset of C++ (apart from the two extra keywords). The language can be partially evaluated and compiled down to GPIR code which can be further compiled to be run on the GPRM.

Methods and techniques that can be applied to interpret and partially evaluate purely functional code have also been explored.

This project also leaves a lot of opportunities for future work. Compilers and Programming Languages in general have multiple areas where there are constantly improvements to be made, most notably in language features and optimizations for code generation. A couple of possible improvements are explored in Chapter 5.

```
1 /* TBB */
2 class Sort: public task {
3 public:
4  int*A; int*tmp; int size; int cutoff;
5  Sort(int*_A, int*_tmp, int _size, int _cutoff):
6   A(_A), tmp(_tmp), size(_size), cutoff(_cutoff){}
7  task*execute() {
8   if (cutoff==1) {
9     SeqSort(A, tmp, size);
10  } else {
11    Sort& a = *new(allocate_child()) Sort(A, tmpA,
          half, cutoff/2);
12    Sort& b = *new(allocate_child()) Sort(B, tmpB,
          size-half, cutoff/2);
13    set_ref_count(3);
14    spawn(a);
15    spawn_and_wait_for_all(b);
16    Merge& c = *new(allocate_child()) Merge(A, B,
      tmp, size);
17    set_ref_count(2);
18    spawn_and_wait_for_all(c);
19  }
20  return NULL;}
21 };
22
23 //The other three approaches use the function Sort
24 void Sort(int*A, int*tmp, int size, int cutoff){
25  int half = size/2;
26  int*tmpA = tmp;
27  int*B = A + half;
28  int*tmpB = tmpA + half;
29
30 /* OpenMP */
31  if (cutoff==1) {
32    SeqSort(A, tmp, size);
33  } else {
34   #pragma omp task
35    Sort(A, tmpA, half, cutoff/2);
36   #pragma omp task
37    Sort(B, tmpB, size-half, cutoff/2);
38   #pragma omp taskwait
39   #pragma omp task
40    Merge(A, B, tmp, size);
41   #pragma omp taskwait
42  }
43
44 /* Cilk Plus */
45  if (cutoff==1) {
46    SeqSort(A, tmp, size);
47  } else {
48    _Cilk_spawn Sort(A, tmpA, half, cutoff/2);
49    _Cilk_spawn Sort(B, tmpB, size-half,cutoff/2);
50    _Cilk_sync;
51    _Cilk_spawn Merge(A, B, tmp, size);
52    _Cilk_sync;
53  }
54 }
```

Figure 4.2: Mergesort implementations in TBB, OpenMP, and Cilk Plus [21]

Chapter 5

Future Work

5.1 Configuration file generation

The GPRM framework uses YAML configuration files[22] when compiling the GPIR code. An example file for the MergeSort implementation is as follows:

```
# Core Services Configuration
System:
  Version: 3.0
  Libraries: [MergeSort, CoreServices]
  NServiceNodes: 241 # excluding gateway; this is actually the number of threads
  ServiceNodes:
    ctrl: [ 1, [CoreServices.SEQ ] ]
    control: [ 1, [CoreServices.BEGIN] ]
    a: [ 2, [CoreServices.ALU] ]
    reg: [ 2, [CoreServices.REG ] ]
    MS: [2, [MergeSort.MergeSort] ]

  Aliases:
  # Alias Name (case sensitive): FQN
    begin: control.CoreServices.BEGIN.begin
    par: ctrl.CoreServices.BEGIN.begin
    seq: ctrl.CoreServices.SEQ.seq
    'reg.write': reg.CoreServices.REG.write
    'reg.read': reg.CoreServices.REG.read
    'reg.inc': reg.CoreServices.REG.inc
    'MS.array': MS.MergeSort.MergeSort.array
    'MS.serial_ms': MS.MergeSort.MergeSort.serial_ms
    'MS.merge_two': MS.MergeSort.MergeSort.merge_two
# These used to be "ALU_names"
    '+': a.CoreServices.ALU.plus
    '*': a.CoreServices.ALU.times
```

The YAML file defines what libraries and services are required to be compiled, number of threads, and aliases for functions in the GPIR code.

It is entirely possible to generate this information from the GPC compiler rather than manually having to create these files.

5.2 Language Features

There are many ways the GPC language can be extended to have more functionality. A couple of notable features are:

- C++11 lambdas - giving the language the ability to implement higher order functions.
- Templates - giving the ability to use some C++ Standard Template Library structures such as Vectors.
- Header/Module system - currently a GPC source file has no way of including GPC code from other files.

5.3 Compiler Optimizations

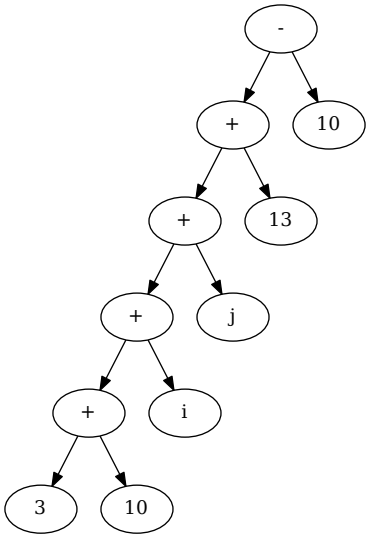
There are a few optimizations that can be performed by the compiler to reduce the work needed to be performed by the GPRM at runtime.

5.3.1 Binary Expression Reduction

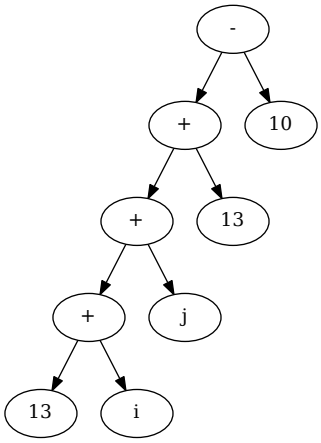
Given the following GPC code:

```
1
2  GPRM::Kernel::Test test;
3
4  void binaryOp() {
5      seq {
6          int i = test.m1();
7          int j = test.m2();
8          test.m2(3 + 10 + i + j + 13 - 10);
9      }
10 }
```

The expression within the `test.m2` method is read in as a binary expression tree which can be represented by the following:



The tree above once evaluated transforms into the following tree:



We can see this reduction in the generated GPIR code:

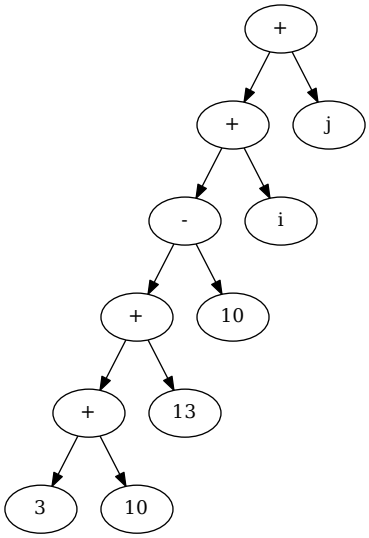
```
;binaryOp.yml

(seq
    '(reg.write[0] '1 test.m1[0])
    '(reg.write[0] '2 test.m2[0])
    '(test.m2[0]
        (-[0]
            (+[0]
                (+[0]
                    (+[0] '13 (reg.read[0] '1))
                    (reg.read[0] '2))
                '13)
            '10)
        )
    )
```

This shows the limitations of the expression evaluation explained in section 3.6.8. Once a value that can't be worked out at compile time is met, it is not possible to evaluate expressions any value further up the tree. In the given example it is clear that it is possible to evaluate the expression further.

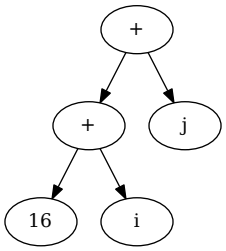
One way to improve the evaluator would be to transform the tree before evaluating it. Using the axioms of the binary operations in the tree (e.g. addition being associative) and the type of values in the leaf nodes, it should be possible to rearrange the nodes in the tree to create a new tree which represents an expression equivalent to the expression represented by the starting tree. This method is also known as **Algebraic Simplification and Reassociation of Addressing Expressions**[23, Section 12.3]

The specific details of how the tree transformations work and implementation into the compiler is left as future work.

In this example, one "optimal" transformation of the tree would result in the following tree:



The evaluator would then reduce this tree down to the following tree:



This generates much simpler GPIR code:

```
1  ;binaryOp.yml
2
3  (seq
4      '(reg.write[0] '1 test.m1[0])
5      '(reg.write[0] '2 test.m2[0])
6      '(test.m2[0]
7          (+[0] (+[0] '16 (reg.read[0] '1)) (reg.read[0] '2)))
```

5.3.2 Pure Function Reduction

When a function is pure (i.e. it contains no method calls on kernel objects, and all arguments given to it when it is called can be evaluated at compile time) the function call should be able to be evaluated down to a single constant value.

The following GPC code is used as an example:

```
1
2  GPRM::Kernel::Test test;
3
```

```
// Pure Function
int f() {
    int a = 5;
    int b = 6;
    int c = 10;
    int d = 22;
    return ((a * b) / c) & d;
}

void pureFun() {
    seq {
        int a = f();
        test.m1(a);
    }
}
```

The function `f` does not call any methods on kernel objects, and has no arguments. Every time `f` is called it returns the integer value "2". Therefore wherever `f` is called can be replaced with the integer value "2". However the compiler generates the following GPIR code for this example:

```
;pureFun.yml

(seq
    '(reg.write[0] '1 '2)
    '(test.m1[0]
        (reg.read[0] '1)))
```

When `f` is called its value is stored in a register. This is currently how function calls store their return value when evaluated. An improvement to function evaluation to support reduction of pure functions would result in generating this simpler GPIR code:

```
;pureFun.yml

(seq
    '(test.m1[0] 2)
)
```

5.3.3 Register Tracking

As explained in section 3.6 register tracking is non existent and every time a new value needs to be stored the register counter is incremented. After a while this can possibly use a lot of memory. A more efficient method would be to track registers being used and once a variable goes out of scope then the register associated with that variable gets "freed".

The "free list" can be implemented in Haskell as a simple linked list containing the number of registers that are free. Every time a register is "freed" its number gets added to the tail of the queue. When attempting to assign a variable to a register the list is checked to see if it has any free registers in it. If it has any then the head of the queue is taken as the register number. Otherwise the total register counter is incremented and the register counter is used as the register number for the new variable.

For each scope the register numbers allocated are kept track of. Once the interpreter goes out of a scope then every register number allocated in that specific scope is added to the "free list" as those values will not need to be used again. This method of register tracking and freeing will use far less total registers than using a new register for each new variable.

Bibliography

[1] A. Tousimojarad and W. Vanderbauwhede, "The Glasgow Parallel Reduction Machine: Programming Shared-memory Many-core Systems using Parallel Task Composition," *Electronic Proceedings in Theoretical Computer Science*, vol. 137, pp. 79–94, 2013.

[2] C. E. Leiserson, "The Cilk++ Concurrency Platform," *The Journal Of Supercomputing*, vol. 51, pp. 244–257, March 2010.

[3] "Cilk Plus FAQ." `https://www.cilkplus.org/faq`. Accessed: 2015-03-15.

[4] "Cilk Plus Tutorial." `https://www.cilkplus.org/cilk-plus-tutorial/`. Accessed: 2015-03-15.

[5] "Cilk Plus Array Notation." `https://www.cilkplus.org/tutorial-array-notation`. Accessed: 2015-03-15.

[6] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[7] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The Design of OpenMP Tasks," *IEEE Transactions. Parallel Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2009.

[8] A. Kolosov, E. Ryzhkov, and A. Karpov, "32 OpenMP traps for C++ developers." `https://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers`. Accessed: 2015-03-15.

[9] C. Pheatt, "Intel® Threading Building Blocks," *J. Comput. Sci. Coll.*, vol. 23, no. 4, pp. 298–298, 2008.

[10] G. Contreras and M. Martonosi, "Characterizing and Improving the Performance of Intel Threading Building Blocks," in *Proceedings of the 2008 International Symposium on Workload Characterization*, 2008.

[11] "Intel Thread Building Blocks Benefits." `https://software.intel.com/en-us/node/506043`. Accessed: 2015-03-15.

[12] K.-F. Faxén, "Wool-a work stealing library," *SIGARCH Computer Architecture News*, vol. 36, no. 5, pp. 93–100, 2009.

[13] M. Davis, *Computability & Unsolvability*. Dover, 1958. ISBN 978048661471.

[14] "Cilk Plus *cilkfor* documentation." `https://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-ABF330B0-FEDA-43CD-9393-48CD6A43063C.htm`. Accessed: 2015-03-15.

[15] D. Leijen and E. Meijer, "Parsec: Direct Style Monadic Parser Combinators for the Real World," Tech. Rep. UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[16] "The Haskell Cabal." `https://www.haskell.org/cabal/`. Accessed: 2015-03-11.

[17] "HUnit website." `http://hunit.sourceforge.net/`. Accessed: 2015-03-18.

[18] "Haskell Program Coverage toolkit user guide." `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/hpc.html`. Accessed: 2015-03-18.

[19] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 2, pp. 181–210, 1991.

[20] M. M. Brandis and H. Mössenböck, "Single-pass generation of static single-assignment form for structured languages," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1684–1698, 1994.

[21] A. Tousimojarad and W. Vanderbauwhede, "Number of Tasks, not Threads, is Key," *in Parallel, Distributed, and Network-Based Processing (PDP). IEEE 23rd Euromicro International Conference on. At Turku, Finland*.

[22] "YAML website." `http://yaml.org/`. Accessed: 2015-03-18.

[23] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN 1558603204.