Andrei-Vlad Bădeliță

Implementing Parallelism in Lisp for REDUCE

Part II Computer Science Dissertation

Trinity College

April 25, 2019

Declaration of originality

I, Andrei-Vlad Bădeliță of Trinity College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Andrei-Vlad Bădeliță of Trinity College, am content for my dissertation to be made available to the students and staff of the University.

Signed [signature]

Date [date]

Proforma

Candidate Number:	**2320D**
Project Title:	**Implementing Parallelism in Lisp for REDUCE**
Examination:	**Computer Science Tripos – Part II, 2019**
Word Count:	**11232**
Final code line count:	**8803**
Project Originator:	Dr. Arthur C. Norman
Supervisor:	Dr. Arthur C. Norman

Original Aims of the Project

To implement a multi-threaded version of a Lisp interpreter. It should allow the user to create threads and offer the primitives necessary to write parallel programs. The original language is able to build and run a complete computer algebra system, which the parallel version should be able replicate fully. Additionally, a number of tests and algorithms will be implemented to evaluate correctness and the potential benefits of parallelism.

Work Completed

All of the original goals have been achieved. The multi-threaded Lisp is showing predictable speed-ups when executing tasks in parallel, while preserving full backwards compatibility with the original language.

Special Difficulties

There were no special difficulties encountered.

Contents

Chapter 1

Introduction

The motivation for this project is to explore the implementation of multi-threading capabilities within a working compiler and assess the benefits and trade-offs it brings to a real-world application with a large, actively-developed body of code.

1.1 REDUCE

REDUCE [1] is a portable general-purpose algebra system (CAS). It enables symbolic manipulation of mathematical expressions and provides a wide range of algorithms to solve problems spanning many fields, with arbitrary precision numerical approximation. It has a friendly user interface and can display maths and generate graphics.

REDUCE is one of a few open-source general-purpose CAS programs, alongside Maxima and Axiom. The three projects are all built on top of different Lisp kernels. At the time of this writing, none of these projects have any multi-threaded capabilities. My aim is to remove this limitation for REDUCE. The project is using its own Lisp dialect which is similar to Common Lisp, but has its own design and set of capabilities.

1.2 VSL

Over the years, there have been multiple implementations of the Lisp compiler REDUCE is using, with different goals: PSL, CSL and VSL. VSL is an interpreted language written in the programming language `C`. It is fully capable of building the entirety of REDUCE, supports all the major platforms and architectures, and is well optimised for speed, minimising the performance tradeoff of being interpreted.

1.3 Benefits of multithreading

The idea of using parallel computing to speed up computer algebra computing has come up in research papers for many years [3, 7, 6], but much of the activity now pre-dates the now ubiquitous multi-core CPUs used in modern computers and the amount of memory they now provide. Moreover, advancements in single-core CPU performance has slowed down significantly, as clock speeds have stagnated and even gone down in recent years.

The biggest area of improvement in these new CPUs is their core count and number of hardware threads. Binding the performance of Reduce to single-threaded performance is likely to lead to a limitation in speed gains from new hardware.

1.4 Code examples

To help explain the concepts I introduce, I will use code fragments showing the algorithms. These fragments will be a simplification of the original code, in order to remove the need for context within the rest of the codebase. To that end, the names and function interfaces are different from the real implementation. More complete versions of the code fragments I showcase can be found in the Appendix.

For examples relating to the implementation of the interpreter, I will be using C++. For work related to my evaluation, I will be showcasing the ParVSL language, which is a type of Lisp. Reduce implements an additional syntax on top of Lisp, called RLisp. I have tried my best to keep RLisp examples short and as readable as possible. Where I think the syntax of the language get in the way of readability, I switch to pseudo-code to aid my explanations.

Chapter 2

Preparation

2.1 ParVSL

I have forked the the original VSL project into a new language which I call for simplicity Parallel VSL, or ParVSL. ParVSL is fully backwards compatible with VSL and will be tested against it for performance.

VSL was a good candidate for this project because it featured a complete, working Lisp implementation while being small enough to be a manageable project. The entire VSL codebase consists of around 10000 lines of code, all of which was originally contained within a single file. Before making changes in critical areas, I spent some time familiarising myself with the code. This included splitting the project in more more files, fixing a few obvious bugs, and porting to C++.

2.2 From C to C++

The VSL language was written in the programming language C. C is a language with no standardised multi-threaded model and no native support for multi-core programming. Furthermore, it has no well-defined memory model, and no defined ordering of memory accesses. Multi-threaded programming is only possible in C through a third-party library, such as `Windows threads` or the the `POSIX threads` library on UNIX systems. Support further has to be guaranteed by individual compilers and operating systems and can break between versions.

C++ is a superset of C and can compile existing, standard C code easily. The C++11 standard addresses the above omissions, making C++ a multi-threaded language. While in some cases the implementation uses the same libraries as the C equivalent (e.g POSIX), we do not have to think about these details and the code we write is fully portable. The only requirement is that a C++11 compliant compiler is used to compile the code and then which platforms these compilers can target. As of today, the C++11 standard has matured enough that all the large compiler vendors (i.e GCC, Clang, Visual C++, Mingw, etc.) fully support it on the major platforms (e.g x86, ARM and SPARC).

The first change I have made to the implementation is to clear it of any incompatible code and compile it with a C++ compiler. This was a trivial task and mostly involved

adding a few more explicit casts. However, I have been slowly transitioning the code from idiomatic C++ as I analysed more parts of it and became confident those changes wouldn't affect the semantics of the program.

2.3 Throughput vs latency

When optimising for performance in a programming language, we have to analyse the tradeoff between total throughput and latency. Optimising for latency means minimising the duration of any individual task in the program, and increasing availability. Optimising for throughput involves minimising the total running time of the program. For example, a web server would benefit more from reducing latency of any individual request.

In a CAS program the user is most likely to care about throughput, i.e. compute the output of large problem sizes as quickly as possible. The program is single-user and has a simple interface. The only case for low latency is in the responsiveness of the graphical user interface. This is already provided by the operating system so our main goal is directed towards minimising throughput in the application. This is particularly important when designing the garbage collector.

2.4 Memory allocation

Memory is managed by the interpreter. It allocates a large block of memory at the beginning, which it then manages as a contiguous array. When running out of memory, an extra block of the same size as current available memory is malloc-ed, doubling the amount available. These blocks are never freed until the end of the program. They are sorted by their pointer locations, and carefully *joined* together to maintain the abstract model of contiguous memory. Binary search is used to identify the block containing a location.

2.5 Garbage collector

An important feature of Lisp languages is their garbage collectors. Garbage collectors allow the programmer to design code without having to worry about the lifecycle of their data, the internal memory model and managing pointers. This makes Lisp code significantly easier to write, leaving the burden of providing safety and efficiency to the compiler.

In effect, the garbage collector is an important component of the VSL interpreter and careful considerations have to be made when modifying it. First of all, any bugs in the garbage collector may leave the memory in an invalid state, corrupting the state of the program and leading to undefined behaviour in C. Such errors are also very difficult to spot and debug, as they can go undetected until the particular region of memory is accessed again.

2.5.1 Cheney's algorithm

The approach a garbage collector uses to deal with freed memory affects both its performance and memory usage. Before the first garbage collection cycle, memory can simply be allocated in a continuous fashion, making it compact and fast. When the garbage collector finds unreachable objects and eliminates them, they will leave *gaps* behind and cause *fragmentation*. Not dealing with fragmentation leads to wasted memory. Keeping track of the gaps and filling them with objects of the right size involves extra book-keeping which can be quite expensive. Ultimately, it is impossible to guarantee the gaps are filled efficiently, because the garbage collector cannot predict future memory allocations, and thus heuristics have to be employed.

VSL avoids this problem entirely by using a copying garbage collector. This means it compacts memory by moving traceable objects to a new region. The unreachable objects are simply not moved and they will eventually be overwritten. This method has the advantage that it fully compacts memory, fixing the issue of fragmentation in an efficient, straight-forward way. The main trade-off this approach has is the total memory usage. A region of memory at least as large as the one in use has to be used to copy the live objects into. In addition, when a very large amount of memory is in use, the copying of all live memory might become more expensive than managing the free memory. Finally, this approach is not incremental. The entire heap is scanned and cleaned every run. Other ways might reduce latency by collecting only partially in more, but shorter cycles.

The problem sizes in REDUCE are usually not bound by memory on modern computers, with none of the applications being known to use more than one gigabyte. Its application is also not latency-sensitive. It is used for solving large problems as fast as possible. A large compaction stage is more efficient as it reduces time switching between running code and collecting garbage and minimises fragmentation. This makes the copying approach suitable for the language.

Cheney's algorithm [2] is a method of stop-the-world copying garbage collection. The virtual *heap* is divided into two halves, and only one half is in use for allocations. The other half is considered free and used during garbage collection. When the first half is full and garbage collection is triggered, all traceable objects are copied over to the second half. Then, the two halves are swapped.

To start the tracing we need to consider a root set: a subset of object which are known to be accessible. One example of elements in the root set is the table of symbols which are defined at the start of the garbage collection. The stack will also contain pointers to objects and must be scanned when computing the root set. While these are the main components of the root set, the interpreter may contain others depending on language features and implementation, all of which must be spotted and added. The root set must be conservative as missing any object which should be in the root set will result in collection of live objects, invalidating the program. It is always preferable to over-approximate the root set.

Once we have distinguished a root set, we can trace all references to build the reachable set. Objects may contain references to other objects, which are also considered reachable. For example, all elements of a list must be traced recursively. The reachable set is the

transitive closure of the reachability relation. All objects in the reachable set must be kept during collection, while everything else may be safely collected.

The algorithm presented above is a heavily simplified abstraction. The root set includes other locations apart from the symbol table, and all of them have to be handled carefully. This was an area of particular importance when adding multi-threading.

The `copy` and `copycontent` procedures have to read the heap to determine the type, size and fields of the `LispObject`, and act accordingly. This approach of storing all the information inside the virtual heap and manually accessing it as needed has significant performance benefits. `LispObject` becomes a simple aliasing for a pointer type, it allows many different types of objects (integers, floats, strings, lists) to be accessed in a unified way, while staying compact in memory. The disadvantage is that it is is difficult to track memory corruption, making debugging more difficult. This will become a problem if any bug in the code produces a data-race.

Please refer to appendix A.1 for a more detailed implementation of the algorithm.

2.5.2 Conservative GC

One important design aspect of calculating the root set is how to handle references on the stack. Garbage collection may start in the middle of a large computation and the references on the stack cannot be discarded. One safe, but slow approach of dealing with this is to keep a virtual stack. Such a stack could be well typed and easily scanned to find references. However, it would be much slower by adding a level of indirection to each expression, and it would also make the code more difficult to manage.

Another approach is to tag words in memory. This approach is used, for instance, by the OCaml compiler, where the least significant bit is a tag bit, indicating whether that word is a pointer reference or just data. This approach is better than the virtual stack, but has the drawback of limiting the integer types (e.g 63bit vs 64), and requiring additional instructions (i.e shifts) to do arithmetic.

The approach VSL uses is to be conservative. It treats all values on the stack as potential references, called *ambiguous* roots. This means we are overestimating the set of roots. Unlike *unambiguous* roots (like the symbol table above), we have to be careful when handling these values, and cannot manipulate them as `LispObject`. This rules out calling `copy` or `copycontent` above on them, but they still need to be kept during garbage collection. The solution was to *pin* them. Any location on our heap which is pointed to by an ambiguous root is pinned and not copied over. Additionally, the `allocate` function will have to check for pinned items on the heap and skip over them. This additional bookkeeping is manageable, When building the entirety of REDUCE, the number of pinned items is never larger than 300. Considering memory used is in the order of megabytes, these pinned locations cause negligible fragmentation.

2.6 Symbols and variable lifetime

As the original language is decades old, its mechanism for variable lifecycle is not in line with that of modern languages. This mechanism was counter-intuitive at first, and is

lacking in providing safety to the user of the language.

There are two lifetime specifiers for symbols: *global* and *fluid*. It is important to note that they do not refer semantically to variables but only to **symbol names**.

A *global* symbols has only a single globally visible value. That means you cannot bind the name to any local variable. For instance if `x` is declared global, it then cannot be used as a function parameter name, or in a let binding.

A *fluid* variable has a global value, but can also be locally bound. Fluids behave more like globals do in other languages, allowing the name to be reused.

`Let` bindings and function parameters introduce *local* symbols. If the symbol name is already declared global, it will result in an error. If it is a fluid and has a global value, that value will be shadowed for the lifetime of the binding.

To make it easier to use it as a scripting language, Reduce's Lisp allows using symbols that aren't global, fluid, or locally bound. These will act like local variables, except their lifetime will be defined for the duration of the program. For single-threaded programs, this distinction is not important: these variables would act like fluids. However, when implementing multithreading, fluids have a global value visible to all threads, while these symbols are only visible on their local thread. I will call these symbols *unbound locals*.

2.7 Saving state to disk

REDUCE has an important feature which allows the user to preserve the state to disk. A `preserve` instruction can be used to do so, and the user is able to specify a function to run on restart. Preserve saves the entire state of the program, including memory and symbols. The feature involves careful manipulation of the world state and it can maintaining it unaffected when enabling multi-threaded programs is complicated. I ran into a few challenges when implementing ParVSL, which I will discuss in the implementation section.

Chapter 3

Implementation

3.1 Integrating threads

After familiarising myself with the VSL codebase I tried to implement the simplest form of multi-threading and test out how the language would behave. I added a new function to VSL which would takes a piece of code as an argument, starts a new thread and joins again with the main thread. Once implemented, I could run first and simplest unit test for ParVSL:

```
> (dotimes (i 4) (thread '(print "Hello World")))
"Hello World"
"Hello World"
"Value: Hello Worln"Hello Woird"
ld"
```

We can immediately observe the effects of parallelism in action. At this point, the code above is just about the only working example. The interpreter is not thread-safe and data races on global variables (including printing to the same stream) lead to undefined behaviour. Printing still seems to work, however most other functions would fail. The spawned threads can only handle strings and will crash on handling numbers or symbols, or when trying to allocate anything. There is no inter-thread communication, exception safety, and any garbage collection would produce a segmentation fault. To be able to write a more complicated test, I need to make changes in all ares of the compiler.

To manage the threads, I made further use of the C++11 standard library. I created a global hashmap storing all the running threads. Creating and joining a thread is done under a mutex lock. Each thread is assigned its own unique identifier, which is returned to the user. The user can then use the identifier to join the thread.

3.1.1 RAII classes

The RAII (Resource Allocation Is Initialisation) design pattern is common in C++. It is a programming technique which binds the lifetime of resources to an object's lifetime. Normally, C and C++ are manually managed, meaning all resources have to be carefully

tracked by the programmer. This makes it easy to introduce bugs when there is an attempted access to an unallocated resource, or leaks when a resource is not released after use. C++ classes offer a solution to this problem. Classes have both constructors and destructors, and these are automatically called when the object is created and when it becomes unavailable, respectively.

```
{
  // obj1 is constructed here, unlike Java,
  // where it would be a null reference.
  Foo obj1;
  Bar obj2;
} // End of scope, obj2 is destructed, followed by obj1.
Foo *obj3 = new Foo(); // Foo constructor called here.
delete obj3; // Foo destructor is called here on obj3.
```

We can use this mechanism to implement automatic resource management, or RAII. We simply make sure the underlying resource is allocated in the constructor and deallocated in the destructor. Smart pointers like `std::unique_ptr` are a good showcase of the power of RAII. As soon as the smart pointer objects become inaccessible (e.g by going out of scope), the underlying pointer is safely deleted, providing a primitive (but very effective) form of automatic reference counting.

When changing the codebase to use C++ features I found various opportunities to apply the RAII pattern, when managing threads, shallow binding and garbage collection.

Thread objects in C++ are not implemented in a RAII style in the standard library. When a thread object is destructed, it must be in an *unjoinable* state. A thread is unjoinable if it either has joined or it has been detached. If an unjoinable thread is ever destructed it will cause the entire C++ program to terminate.

I have wrapped all threads in a `ThreadRAII` class, so that whenever they go out of scope they are automatically joined (when still in a joinable state). This guarantees there will be no premature termination in the case the user does not handle the thread correctly and that the ParVSL interpreter will exit cleanly.

3.2 Storage management

All the memory is global and shared, and multiple threads will often try to allocate concurrently, causing contention. A naive solution to this problem would have been to use a mutex lock on allocations. While this would be easy to implement, it would also severely degrade performance. Locks are expensive even in single-threaded scenarios with no contention, and allocations are very common when evaluating Lisp. This is especially the case in an interpreted Lisp were no allocations are optimised away. Simply reading the code to evaluate allocates $O(n)$ times for code of length n. That is because code in Lisp code is data. Each instruction is a list data structure, with each elements allocated. Serialising allocations is guaranteed to slow down the language enough to cancel any advantage multi-threading brings to the language.

3.2.1 Memory allocation

I wanted to allow multiple threads in parallel without affecting the performance of allocations. To do this I had to use a lock-free approach. To do this, I further split the memory into regions, which I call *segments*. A segment is a thread-local region for allocation in memory.

Just like before, memory is allocated to the segments in a continuous fashion. A pointer indicates the start of the non-allocated part of the segment (the *segment fringe*.), while another tells us the end of the segment (the *segment limit*).

Now, contention is reduced to getting a new segment. Each thread only allocates within its own segment, so allocations do not require any synchronisation, and they still only require incrementing one variable in most cases. If the requested allocation would bring the fringe over the segment limit, then the current segment is *sealed* and a new one is requested.

I carefully modified all the areas where allocations are preformed to use the segment fringe instead of the global fringe. The global fringe is only moved to assign a new segment. Writes to the global fringe are executed under a mutex lock, while the segment fringe is a thread local variable accessed without any locks.

If the requested memory is larger than the segment size (e.g a large string or number), the current segment is sealed, then the large object will occupy its own custom segment, and the thread will then have to request a new segment.

There is a trade-off involved when choosing the segment size. If the size is too small, there will be a lot of contention on requesting segments, leading back to the original problem of locking on every allocation. If the segment size is too large, there is a risk of threads holding large amounts of memory without using it and causing early garbage collection cycles. This is because reclaiming memory is requested when a new segment cannot be created, regardless of how much free space there is in existing segments. While trade-off depends on the total memory, I have found a good compromise for segments of a few kilobytes each.

3.3 Garbage collection

I will use the shortened form *GC* to refer to garbage collection throughout the rest of this chapter for brevity.

The *garbage collector* (GC) has to account for the state of all threads. These threads have to be synchronised and in a safe state to initiate the GC. They must also be included in the calculation of the root set.

I store all the thread-local information required for synchronisation in a class called `Thread_data`. This class is populated when a thread is started and updated before and after a GC cycle. All threads register themselves in a global thread table at start-up. The thread starting the GC can use this table to check the status of the other threads.

The *ambiguous root set* is the set of potential references found on the stack. The stack is a continuous area of memory to managed by incrementing and decrementing a stack pointer. The stack pointer is normally not accessible from C++, however we can

dereference a stack variable to find its location on the stack. VSL remembers the position of the stack at the beginning of the execution and calculates this again before garbage collection. Then it scans the all locations in between for ambiguous roots.

```
uintptr_t C_stackbase;

int main() {
  // t is any variable on the stack, before execution of VSL code begins
  int t;

  // When starting VSL, we store the base of the stack
  // Here we need to cast the reference to an in type
  // then align to the size of a LispObject
  C_stackbase = ((intptr_t)&t & -sizeof(LispObject));
}

void garbage_collection() {
  int t;
  // before GC, we note the head of the stack
  uintptr_t C_stackhead = ((uintptr_t)&t & -sizeof(LispObject));

  // scan the stack from its head to base (the stack grows downwards)
  for (uintptr_t s = C_stackhead;
       s < C_stackbase;
       s += sizeof(LispObject))
  {
    if (in_heap(s)) { // check if s points within our virtual heap
      // we found an ambiguous root
      set_pinned(s);
    }
  }

  inner_garbage_collection();
}
```

Each thread will have its own stack, so I had to modify the code to scan all the stacks before garbage collection. This is one reason I had to pause work on all threads for GC. If I hadn't, then a thread might change add references to heap objects on its stack after those objects have been marked safe to delete.

When a thread is initialised, I save its own stackbase in `Thread_data`, and then also save its stackhead when it is paused to wait for GC. All these stack ranges are scanned before I start garbage collection:

```
void garbage_collection() {
  for (auto thread: thread_table) {
```

```
    // scan the stack from its head to base (the stack grows downwards)
    for (uintptr_t s = thread.C_stackhead;
         s < thread.C_stackbase;
         s += sizeof(LispObject))
    {
      if (in_heap(s)) { // check if s points within our virtual heap
        // we found an ambiguous root
        set_pinned(s);
      }
    }
  }

  inner_garbage_collection();
}
```

3.3.1 Garbage collection locks and safety

The garbage collector is *stop-the-world*. The thread initiating garbage collection must wait for all threads to be ready. Similarly, all threads must regularly check whether a garbage collection cycle was initiated and act accordingly.

The first idea I had was to trap all calls to allocate memory and check whether garbage collection is needed. To do this, I could simply reset all thread segments. Threads would need to allocate eventually and would request a new segment, at which point the would need to call the GC. This solution is incomplete however. First of all, a thread might be busy for a long time without needing to allocate. This would cause all other threads to be idle waiting for it to finish.

A bigger issue was the risk of deadlocks. If thread A is waiting for the result of a computation on thread B, but thread B was paused waiting for the GC, then the program is deadlocked. Similarly, any effectful computation, like waiting for user input will prevent the collection from starting.

```
// global variables for synchronising garbage collection
std::atomic_int num_threads(0);
std::atomic_int safe_threads(0);
std::condition_variable gc_wait_all;
std::condition_variable gc_cv;
std::atomic_bool gc_on(false);
```

To deal with the issue of blocking calls, I defined another state threads can be in: *safe for GC*. A thread is in a safe state if it has saved all the information the garbage collector needs to begin (e.g stackbase and stackhead) and guarantees not to run any code that invalidates the garbage collection. Threads go into a safe state whenever they get paused for GC. However, they can also be in safe state when waiting for a blocking call.

I created a special RAII class to handle both scenarios, which I called `Gc_guard`. The class only has a constructor and a destructor and is a way for the thread to promise it is in a safe state.

```
class Gc_guard {
public:
  Gc_guard() {
    int stack_var;
    // save the stackhead
    thread_data.C_stackhead =
      (LispObject *)((intptr_t)&stack_var & -sizeof(LispObject));
    paused_threads += 1;

    // notify the gc thread that this thread is in a safe state
    gc_wait_all.notify_one();
  }
   ~Gc_guard() {
    std::mutex m;
    std::unique_lock<std::mutex> lock(m);

    // wait here for gc to finish
    gc_cv.wait(lock, []() { return !gc_on; });
    paused_threads -= 1;

    // invalidate the stackhead
    thread_data.C_stackhead = nullptr;
  }
}
```

The `Gc_guard` class is accompanied by `Gc_lock`. A thread trying to initiate the GC will have to acquire a `Gc_lock`. The lock will wait until all threads are in a safe state and will prevent other threads from acquiring.

TODO: fix this code

```
class Gc_lock {
  std::mutex m;
  std::unique_lock<std::mutex> lock;
  Gc_guard gc_guard;
public:
  Gc_lock() : m(), lock(m) {
    int stack_var = 0;
    thread_data.C_stackhead =
      (LispObject *)((intptr_t)&stack_var & -sizeof(LispObject));

    paused_threads += 1;
```

```
    gc_wait_all.wait(lock, []() {
      return paused_threads == num_threads;
    });
  }

  ~Gc_lock() {
    paused_threads -= 1;
    gc_on = false;
    thread_data.C_stackhead = nullptr;
    gc_cv.notify_all();
  }
};
```

3.4 Data races in the interpreter

The interpreter is the core of VSL. It runs a read-eval-print loop which makes heavy use of global state and side effects.

All named objects in the lifecycle of the program are *symbols*. All global and local variables, including special ones (like `true` or `nil`) and function arguments are symbols. A global hash-table keeps track of all symbols. This means each name can only be in use in one place at a time.

Lisp is a language with dynamic scope. This has many implications for the interpreter. The following fragment of code is an example of this behaviour:

```
let f x = x + y in
let g () =
  let y = 3 in
  f 2
in
g ()
```

The above example will not compile in any statically scoped languages such as OCaml or C++ because the variable `y` is not defined in its scope. Most dynamic languages, even weakly typed ones, like Javascript or Python, will allow equivalent code as valid, but will encounter a runtime error because `y` is not defined. Lisp, and VSL in particular, has a mach looser concept of scope. In the example above, `y` is defined before the call of `f` and will remain defined until the call to `g` returns.

Shallow binding is the mechanism by which this is achieved. Each symbol is mapped to a single value. When a variable name is bound, e.g. as a function parameter name during a function call or through a `let` statement, its old value is stored by the interpreter and replaced with the new value. At the end of the binding, the old value is restored. The main advantage of this method is performance. Old values can simply be saved on the stack:

```
let varName = expr1 in expr2
```

can be implemented as:

```
void implement_let(string varName, LispObject expr1, LispObject expr2) {
  LispObject symbol = lookup_symbol(varName);
  LispObject oldVal = value(symbol); // store the old value
  value(symbol) = eval(expr1);       // replace with new value
  eval(expr2);                       // evaluate the rest of the code
  value(symbol) = oldVal;            // restore old value
}
```

This mechanism was not designed with concurrency in mind, and is not thread-safe. Many variable names are reused multiple times in a program (e.g `i, j, count`, etc). Multiple threads binding the same variable will override each other's values.

I wanted to fix this while keeping the same dynamic scoping semantics for backward compatibility. One option was to replace shallow binding with functional association lists storing local values, however I was concerned with the performance. Shallow binding has a constant factor, while any associative data structure would add non-negligible overheads to a critical area of the interpreter (symbol lookup).

My approach was to allocate thread-local storage for symbols. Global symbols were unaffected, because rebinding them is illegal in the language. However, for non-globals I used a thread-local array to store the real value, and had the global storage location point to the array location. Each local symbol gets its own unique array index for the lifecycle of the program. Then, each thread will reserve that location within its local array for the symbol. Then, I created the function:

```
// the original value function
LispObject value(LispObject symbol);

LispObject par_value(LispObject symbol) {
  LispObject val = value(symbol);
  if (is_global(symbol))
    return val; // global symbols remain unaffected

  // ASSERT: val is a Location otherwise
  Location loc = value_to_location(val);
  return local_value(loc); // get the thread_local value at that location
}
```

I carefully replaced all calls to `value` to `par_value`. Now, multiple threads accessing the same symbol can do so safely, as they will each access their own versions. This eliminates data races entirely.

3.4.1 Lock-free hashtable for symbol lookup

Just like allocations are a critical region of code in VSL, so are symbol lookups. Every occurrence of a symbol must be looked up in the symbol table. If the symbol does not

exist, it will be created. Multiple threads looking up symbols will cause contention. If two threads try to allocate the same symbol name at the same time, they will invalidate the table.

```
LispObject symbols[SYMBOLS_SIZE];

LispObject lookup(std::string name) {
  size_t loc = hash(name) % SYMBOLS_SIZE;

  LispObject bucket = symbols[loc];

  while (bucket != nil) {
    LispObject s = head(bucket); // first list element

    if (symbol_name(s) == name) {
      // found the symbol
      return s;
    }

    bucket = tail(bucket); // rest of the list
  }

  LispObject s = allocate_symbol(name);
  symbols[loc] = cons(s, bucket);

  return s;
}
```

As before, the naive solution would be to implement a mutex lock on the entire lookup function.

To improve on that I tried to use a reader-writer lock. Reader-writer locks allow grant access to either a single writing thread, or all the reading threads. This would would allow multiple threads to lookup symbols at the same time, however, as soon as one one thread has to create a symbol, all th reader have to wait for it to finish.

Moreover, the lookup function is two-step: first it tries to find a symbol, then it creates one if it didn't find any. If two threads lookup the same symbol at the same time, it is possible for both of them to end try to create it at the same time. The reader-writer lock would not prevent this! It would serialise the writes, so it does prevent undefined behaviour in C++, however it does still create the symbol twice. The pointer to the symbol that the first thread returned from the function would become invalid.

I have found a third approach, based on the Compare-and-swap (CAS) instruction which solves the issue above, while also providing a lock-free implementation. The symbol lookup table is currently implemented as static array of linked list.

Instead, I replaced it with:

```
std::atomic<LispObject> symbol_table[TABLE_SIZE];

LispObject lookup(std::string name) {
  size_t loc = hash(name) % SYMBOLS_SIZE;

  LispObject bucket = symbols[loc].load(std::memory_order_acquire);

  while (bucket != nil) {
    LispObject s = head(bucket); // first list element

    if (symbol_name(s) == name) {
      // found the symbol
      return s;
    }

    bucket = tail(bucket); // rest of the list
  }

  LispObject s = allocate_symbol(name);

  LispObject new_bucket = cons(s, bucket);

  while (!symbols[loc].compare_exchange_strong(
      bucket, new_bucket, std::memory_order_release))
  {
    // search for stored value
    LispObject old_bucket = bucket;
    bucket = symbols[loc].load(std::memory_order_acquire);

    for (LispObject s; s != old_bucket; s = tail(s)) {
      if (symbol_name(s) == name) {
        // Another name created the symbol. Use that.
        return s;
      }
    }

    // Make sure we don't discard new symbols inserted by other threads.
    new_bucket = cons(s, bucket);
  }

  return s;
}
```

3.5 Threads

To start a thread, the user simply needs to call the `thread` function. This function takes as arguments a function name and the argument list and a unique thread id is returned. The function is applied to the arguments in a new thread. The return value is stored until the user call `thread_join` on the thread id, when they can access the respective value.

```
tid := thread('add, {2, 3});
result := thread_join tid;
print result;
```

This allows for simple inter-thread communication, through function parameters and returns. Any function can be run in parallel by simply applying thread on it. This task based approach makes it easy to modify single-threaded algorithms to run in parallel.

My implementation minimises overhead. The list of arguments is managed on the heap which is visible by all threads, so a simple pointer is enough to pass them. For returning values, I must keep track of all unjoined threads and their return values. I do this with a simple hashtable, mapping thread ids to their return values. When a thread returns from its function, I update the hashtable with the returned value. When the user joins a thread, I look up the value in the table, return it, then erase the mapping. All threads have unique ids for the lifetime of the program, so there will never be conflicts in the hashtable. This hashtable has to be thread-safe, and I have implemented it as described in Lock-free hashtable.

I must also be careful to prevent the garbage collector from collecting these parameters and return values. Starting a thread is GC-safe: garbage collection will not begin while a new thread is starting up. This ensures that threads are always in a safe state and registered in the thread-table correctly during garbage collection. At that point, function parameters are tracked just like regular variables so they will be safe from GC. However, return values are stored past the lifetime of their respective threads. To deal with them, I add them to the root set at the beginning of garbage collection.

The handling of both of these root sets has to account for multiple running threads. All the new thread-local storage was added to the unambiguous root set. Additionally, each thread is running its own stack so all the stacks has to be accounted for as the ambiguous root set. The latter is more complicated. All threads have to be paused before garbage collection can begin so that they do not interfere with memory. A simple way to do this is to enable a global flag which all threads check on a regular basis. However, if we are not careful, this can easily cause a deadlock (e.g: thread A waits for a signal from thread B, but thread B is waiting for garbage collection). To solve such issues, I need to make two changes. First, I must modify the functions in the interpreter to poll the global flag. Then I have to make all waiting calls put a thread into a *safe* state before sleeping, so that the garbage collector can proceed with the thread.

3.6 Saving state to disk and reloading

One important feature of the language is the ability to preserve the state of the world at any time and save to disk. It is difficult to keep the same guarantees when multiple threads are running: preserving when some of the threads are running a computation is tricky to define properly. To simplify matter, I have decided that all threads have to be joined before preserving. This way, the state of the world is consistent and relatively easy to restore. I have modified parts of the code to write all the thread-local data back into global storage and then restore it when reloading. This way the same file format is preserved, and I have not broken compatibility between single and multi-threaded images.

Implementing ParVSL, I introduced additional state and book-keeping to manage the extra complexity. When preserving in VSL, all this state is stored to a file by compressing a bit for bit copy of the content. The main use for this is saving the results of computations and function definitions. The REDUCE build is stored in the same way. After building all the necessary files, it preserves the current state inside an image. Loading the image on startup of VSL will effectively start REDUCE.

I wanted to keep the VSL image format in ParVSL so that I would not break compatibility between the two. I note that preserving a running multi-threaded program would be impossible without making major changes to the image format and major redesign of the code. Calling `preserve` also stops the pr

If a thread attempts to preserve while others are running, the contents of the final image would be non-deterministic and generally undesirable, even if I were to resolve all data races. At the same time, I found little benefit to implementing such a feature, as the ability to preserve is useful to retain results of computations, not intermediate states. Saving state to disk is an operation happening at the end of the program.

My implementation on preserve preservation only allows the operation on the main thread, and will wait for all spawned threads to finish running and join.

Chapter 4

Evaluation

To evaluate the project, I worked with existing REDUCE code for regression testing but also wrote my own tests in both plain Lisp and REDUCE RLisp code.

4.1 Single-threaded building of Reduce

My project involved modifying a large body of code and it was almost guaranteed that I would introduce bugs during development. Since the work involves a complex, multi-paradigm programming language, it is not possible to guarantee to cover all possible scenarios that might exhibit new bugs. Thankfully, the language and its direct application are well intertwined, so there is a vey large coverage test already available to demonstrate the most functionalities of the language: building REDUCE.

REDUCE consists of around 400 thousand lines of code, and since VSL was built to run it, all functionality in VSL is being used in REDUCE. In addition to that, REDUCE comes with a comprehensive suite of regression tests, which were written over the years to detect bugs in new code. Finally, almost every library in REDUCE contains a set of tests. These tests involve a large amount of heavy computation, stress testing many different algorithms, using large amounts of memory and requiring multiple cycles of garbage collection.

Between building REDUCE and passing its tests, I can be confident that ParVSL retains backwards compatibility and catch development bugs. Any error while running the code, or any difference in output between VSL is considered a bug. This approach helped me find most of the bugs in my code. The disadvantage was that when I had to debug there was too much code running and it was difficult to pinpoint to origin of the problem.

To aid with this, I had multiple stages to build. The first stage was just building the core, while the second involved building the libraries. If an error showed up while building a library, most other libraries could be skipped to help pinpoint the problem. The problem was not completely fixed as just building the core involved running a very large amount of code, and libraries are also significant in size.

Most of the bugs I could have potentially introduced were data corruption bugs. I used asserts in various places to try to get the program to crash as soon as possible and find

the issue early. Sometimes I had to introduce a system of binary searching the problem by trapping the program early and checking for the state of the computation. I tried to find small tests which would still reproduce the issue, then use the `gdb` [4] debugger to step through the code and try to find the errors. I have also used Valgrind [8], which offers a large set of tools for detecting undefined behaviour and memory corruption.

4.1.1 Bugs in VSL and REDUCE

While debugging I found some discovered issues in VSL. The language hadn't been tested as extensively until I started working on it. These issues were mostly minor. Some functions were not present at all and needed to be reintroduced, for example system functions to get working directory. There were subtle bugs such as wrong hard-coded strings and integers. There were multiple issues with floating point manipulation, again caused by simple human errors (e.g negated condition being checked). Switching between output files would lose buffer content. In one place, a value on the heap was not marked properly, meaning that if a garbage collection was triggered at the right time, it would be collected prematurely, causing corruption. All these bugs have since been fixed.

Reduce itself showed multiple cases of sloppy code. Most of the time, the issue was using a global symbol with a very common name liberally and then behaving weirdly on clashes. The most striking example was when the RLisp interpreter used the symbol `x` parsing, meaning any tests using the variable `x` would have the value be a self-reference to code. Another instance was a name clash in the error handling function, leading to failure to contain exceptions. While these two cases have been manually fixed, there are many other such potential problems within the Reduce packages which will have to be cleaned up.

4.1.2 Benchmarks

In addition to helping me find issues in ParVSL, building REDUCE also provided for a good performance benchmark. The building process simply runs Lisp and RLisp code so running it in VSL and ParVSL will showcase the performance difference between the two in a single-threaded case.

Figure 4.1 show the relative performance of ParVSL compared to VSL. RCORE is the core of REDUCE, skipping non-essential packages from the build. ALG TEST is a set of regression tests normally used to check for bugs in the code. INT TEST tests the symbolic integration package, and provides a good performance benchmark.

A slow-down between 5 and 20 percent can be observed. I have used the GProf [5] tool to profile the code and found the biggest cause of the time difference is the symbol value extraction mechanism described in 3.4. In VSL all symbol values are stored globally with the symbol, whereas ParVSL adds extra book-keeping. The symbol access function is on the critical path, and slows the code by at least five percent. This penalty would easily be eliminated in a compiled language, where the checks for global and fluid variables could be performed statically and optimised away from the runtime. Other causes of the slowdown are thread-local variables in the interpreter, which I discuss in section 4.6.

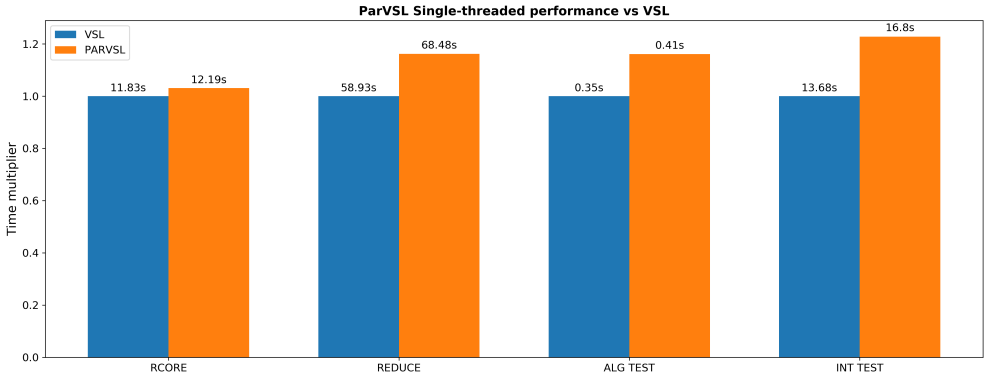


Figure 4.1: Relative performance of ParVSL on single-threaded tests.

4.2 Multi-threading unit tests

While REDUCE provides a large suite of tests for single-threaded behaviour, I had to come up with new tests for multi-threading. Before going into larger examples, I started with a small suite of unit tests in Lisp.

The following test shows multiple threads sharing a variable. They have to acquire a mutex to prevent a data race. All they do is increment it by one, so the final value should be equal to the number of calls.

```
(global '(x x_mutex))
(setq x 0)
(setq x_mutex (mutex))

(de incr_x ()
  (mutex_lock x_mutex)
  (setq x (add1 x))
  (mutex_unlock x_mutex))

(dotimes (i 10000) (thread '(incr_x)))
(print x) % 10000
```

When modifying the garbage collector, I had issues when multiple threads were initiating it, for example I discovered a deadlock with my original GC locks (section 3.3.1), which I then quickly fixed. It was easily reproduced with the following test:

```
% reclaim forces garbage collection
(dotimes (i 16) (thread '(reclaim)))
```

Big numbers use continuous areas of memory and can easily fill in entire segments all at ones. The code below simply raises a running counter to itself in parallel. It helped me discover a that my code was not handling an allocation request being larger than the segment size.

```
% a naive recursive power function
(de pow (a b)
  (cond
    ((zerop b) 1)                  % if b = 0: return 1
    (t (times a (pow a (sub1 b)))) % else: return a * pow(a, b - 1)
  ))

(dotimes (i 1000)
  % raise i^i for really large numbers
  (thread 'pow (list i i)))
```

These tests, among others, were very helpful in finding bugs in the original during development. They were added when testing new functionality, sometimes as a result of finding a bug in a larger example. They now act as regression tests for ParVSL.

4.3 Thread pool

Once I showed that ParVSL could run both single-thread code (i.e build REDUCE) and pass some simple tests for multi-threading, I was able to write more complex code using threads.

Spawning hardware threads directly to parallelise each task can be undesirable. The user has to manage the lifecycle of each thread, making sure to join it and also to manage the number of available threads on the current hardware directly. Failure to do so will quickly result in over-subscription of threads. Each thread object comes with its own overhead including a local stack and operating system handle,

A thread pool is a structure for simplifying parallelism by abstracting away the interaction with hardware threads. A thread pool consists of a work queue for pending jobs and a number of worker threads which execute those jobs as they become available. The number of workers can equal the number of hardware threads so that the program never has to spawn more threads than there available, and threads can be reused. Once a thread pool is created, the user simply needs to submit jobs and they will be automatically parallelised.

4.3.1 A thread-safe queue

The main data structure behind the thread-pool is a thread-safe queue. All threads may push jobs to this queue and all working threads pop tasks from it to execute. Jobs can be executed in any order, and I a queue so that they would be executed in the order they are submitted, which seemed the most natural.

We can implement such a queue easily in ParVSL using a mutex and a condition variable. We start from a simple queue, with the following functions:

- `queue()` creates a new queue
- `queue_push(q, x)` pushes value `x` to queue `q`

- `queue_pop(q)` pops and returns the value at the front of the queue
- `queue_empty(q)` checks if the queue is empty

A $\text{thread}_{\text{safe}}$ queue is simply a wrapper on top of the normal queue:

```
procedure safe_queue();
  {queue(), mutex(), condvar()};
```

We need two procedures: `safe_queue_push(sq, x)` and `safe_queue_pop(sq)`. The latter will wait if the queue is empty until an element is enqueued. The waiting is done using the condition variable:

```
procedure safe_queue_pop(sq);
begin
  scalar q, m, cv, res;
  % unpack the safe queue
  q := first sq;
  m := second sq;
  cv := third sq;

  mutex_lock m;

  while queue_empty q do
    % wait for another thread to push an element and notify
    condvar_wait(cv, m);

  res := queue_pop q;
  mutex_unlock m;
  return res;
end;
```

Now, the push method must notify the condition variable if the queue was empty.

```
procedure safe_queue_push(sq, x);
begin
  scalar q, m, cv;
  q := first sq;
  m := second sq;
  cv := third sq;

  mutex_lock m;
  queue_push(q, x);
  condvar_notify_one cv;
  mutex_unlock m;
end;
```

4.3.2 Managing threads

With the queue implemented we can design the worker threads. The starting thread initialises the queue and starts all the workers as individual threads. It can start either the maximum number of hardware threads (which can be determined using the `hardware_threads()` function), or a custom count. Each thread is passed a reference to the thread pool, so it can access the queue. Once the threads are started, they will only be joined on exit or when the user manually stops the pool.

The mechanism for stopping the queue is a simple atomic flag. Atomics are not offered as a primitive in ParVSL, but can be easily implemented with a mutex lock. There is no direct mechanism for interrupting a thread running a task, but workers can check the flag every time before taking a new task from the queue.

Using the blocking call to wait for the next job mean the worker get stuck and prevent stopping the thread-pool. When the the user tries to stop an empty pool, then all the workers will be in a sleeping state, waiting for the queue condition variable to be notified, causing a deadlock.

```
while atomic_get(run_flag) = 'run do
  // The workers can get stuck here waiting on an empty queue
  job := safe_queue_pop(sq)
  run_job job;
```

Another idea is to not use a blocking call to pop from the queue, but rather spin:

```
while atomic_get(run_flag) = 'run do
  job := safe_queue_try_pop(sq)
  if job then
    // trypop succeeded
    run_job job
  else
    // important to yield here
    thread_yield()
```

This approach solves the issue, but it is important to note the `thread_yield()` call. I have implemented `thread_yield()` to directly call the C++ equivalent. This allows the system to schedule other threads, making sure a waiting worker does not spin the CPU core to 100% until forcefully preempted by the OS. It would also prevent other threads started by the user from doing work.

4.3.3 Waiting for a job's result

In ParVSL, the `thread` function takes another function to execute on the new thread, along with the arguments for that function. The return value of the function call is then recovered when joining the thread with `join_thread`, enabling thread communication.

When switching from threads to jobs in the thread pool, we want to maintain this functionality, otherwise the only way to communicate between parallel jobs would be

through global state, which would severely limit its usefulness. Passing argument for a job is trivial, as they are simply stored in the safe queue, along with the function to be called. However, returning the result of a job required extra book-keeping.

Using the primitives in ParVSL, we can implement a `future` type. A future is a helpful mechanism that allows us to both wait for a task and obtain its return value. A future starts out as empty. It can have any number of readers but only one writer. The writer is usually the creator of the future and will set its value exactly once, at some point after creation. The readers can then try to get the value inside the future. If the future is fulfilled, the get call returns instantly. Otherwise, it becomes a blocking call, waiting until the future is set, then returning the respective value.

Implementing a future is similar to the safe queue, using a mutex and a condition variable. Getting and setting the future requires acquiring the lock. The getter has to wait on the condition variable if the future is not set. The setter notifies all the getters after setting the value. The full implementation can be found in the Appendix.

With the future implemented, we can finish the thread pool, having a mechanism for pushing a job:

```
procedure thread_pool_add_job(tp, fn, args):
  fut := future()
  safe_queue_push(tp.safe_queue, {fn, args, fut})
  return fut
```

The caller can use the future to wait for the result of the job and the workers need to set the future when finishing a job. Finally, I note that the thread pool must deal with exception handling. The worker threads need to catch any error while running the job and report it through the future. Initially, I failed to include it meaning that worker threads unwounded unsafely. This lead to the the thread-pool being unable to signal the thread and fail to terminate. Additionally, threads waiting for the result would also be stuck.

There are many aspects to be considered in the design of a thread pool. I have focused on the main ones, and this thread pool was sufficient for the rest of evaluation. I have successfully used to to parallelise the other experiments in this report. However, depending on the task it could be improved upon with more features. Currently, the number of threads is static, but it could dynamically start and stop threads to accommodate the workload. A more efficient safe queue could be implemented using granular locking. Furthermore, we could reduce contention on the queue by having each worker keep its own queue, and the main queue would act as a dispatcher.

4.4 Implementing Parallel Mergesort

To test the correctness and performance of ParVSL I implemented a few classic algorithms that are relatively easy to parallelise. Sorting is a particularly good example. Mergesort splits a list in two, sort each half recursively, then merges the results to obtain the sorted list. Sorting the individual halves can be done in parallel.

```
tp := thread_pool()

procedure parallel_merge_sort(list):
  if length(list) < 2:
    return list

  xs, ys := split(list)
  sorted_xs_future := thread_pool_add_job(tp, 'parallel_merge_sort, {xs})
  sorted_ys := parallel_merge_sort(ys)
  sorted_xs := future_get(sorted_xs_future)

  return merge(sorted_xs, sorted_ys)
```

We use the thread pool implemented above to achieve parallelism. Without the thread pool, we would have to manually manage threads. Using threads here would have resulting in a new thread spawned for each element in the array. The function would already oversubscribe threads for lists as small as 100 elements. The thread pool only uses a constant number of threads.

4.4.1 Dealing with tasks waiting for other tasks

However, the naive implementation above is incorrect and it will deadlock as soon as the number of jobs exceeds the number of workers. This highlights a shortcoming of the thread pool. In its current state is does not handle tasks enqueuing and then waiting for other tasks. In this case, all the workers will end up waiting for the future (`sorted_xs_future`) without doing any work.

To fix this, I have added extra functionality to the thread pool. An extra procedure `thread_pool_run_job` allows can be called by any thread to run on job on the queue. This procedure is implemented similarly to the worker function, except it only takes at most one job (or none if the queue is empty) from the queue instead of looping. This function should be called by any job which is waiting for another job in the thread pool.

I also needed to implement another function for futures `future_try_get`, which only returns the value in the future if it was fulfilled, without blocking, or indicates failure, without blocking.

Subsequently, I have changed line `10` above to the following code:

```
while null (future_try_get(sorted_xs_future):
  thread_pool_run_job(tp)
```

Now, workers can start (and finish) other jobs while waiting and will not deadlock.

4.4.2 Results

To test the correctness, I simply generate a list of random numbers, then compare the output to that of the sorting function built in Reduce. Afterwards, I could test for performance. I first tuned the parallel version to use the sequential algorithm once if the

	100000	250000	500000	1000000
1	2.351	5.282	11.152	22.840
2	1.563	3.409	5.850	11.706
8	1.319	2.554	4.906	9.054
16	1.071	2.137	3.995	7.514

Table 4.1: Parallel merge sort times by number of workers

list is too small. I found on my machine that around parallelisation became useful once the size of the list was larger than 1000, and I tuned it to a threshold of 5000. Without this optimisation the parallel version would spawn too many jobs ($O(N)$ to be more exact) and the time book-keeping would completely eliminate any benefit of multi-threading. Indeed, it runs an order of magnitude slower on large lists (over 1000 elements).

I have plotted the speed-up boost offered for a number of array lengths, depending on the number of workers.

4.5 Parallel building of Reduce

The entirety of REDUCE is made up of RLISP code, which VSL and ParVSL can simply run to build it. The process can be separated into two steps: building the core of REDUCE, and then building all the additional packages. The core, which I will refer to as RCORE, satisfies most of the dependencies any of the additional packages need, and only takes a fraction of the time to build.

	Time (s)
RCORE	11.83s
REDUCE	59:93s

Once good use of multi-threading would be to speed up this build time by arranging to build individual REDUCE packages in parallel. The RCORE base should still be built sequentially, as it is much more difficult to separate independent tasks. However, the rest of the packages mostly only have RCORE as a dependency.

I extracted a list of 64 packages to build. The starting point was the RCORE image. On VSL, I simply ran the building sequentially:

```
for package_name in packages do
  build_package package_name;
```

The total running time was `14.190`. Then, I used the thread pool to run these builds in parallel:

```
tp := thread_pool(hardware_threads());

pack_futures := {};
for package_name in packages do <<
  pack_future := thread_pool_add_job 'build_package {package_name};
```

```
  pack_futures := pack_future . pack_futures;
>>;

% We need to ensure all jobs are finished.
for pack_future in pack_futures do
  if (future_get pack_future) != nil do
    print "error building package";
```

Running this code will initially fail building any package. This is caused by global side-effects of building the packages. The first issue is that all packages use global symbols with common names for storage. Any global symbol used by a package should use a globally unique name, usually by prepending the package name to the symbol name. Unfortunately, the writers of these packages didn't always follow good practices. It is outside the scope of this project to fix all Reduce packages, however I tried to work my way around that.

I modified the ParVSL code so that when I enable a compilation flag it tracks all reads and writes to global values of fluid or global symbols. Then, I wrote a python script which builds each package individually with the build flag on and saved those accessed to a file. The script then computed all conflicts between packages. The script then generated a Reduce test file which would remove all global access on building the packages:

```
% Force conflicting symbols to be fluid.
% Here, global1, global2, etc. are the names of those conflicting symbols
fluid '(global1 global2 ...)
fluid '(store!-global1 store!-global2 ...)

store!-global1 := global1;
store!-global2 := global2;
...

procedure build_package_safe(name);
begin
  % we bind the conflicting variables locally
  scalar global1, global2 ...;

  % restore their original global values inside the local scope
  global1 := store!-global1;
  global2 := store!-global2;
  ...

  build_package(name);
end;
```

The generated file makes use of the dynamic scoping mechanism in Lisp. When building the package with `build_package_safe`, all accesses to those global symbols will in-

stead access locally bound ones. This removes all conflicts between symbol accesses, without needing to modify code in the Reduce packages themselves.

The second issue I found was that any package can change the access specifier of a global symbol. One package could make a symbol global, while another tries to bind it locally causing an error. This issue could not be solve at the Lisp level, so I had to add another temporary flag to the ParVSL interpreter. Effectively, I disabled globals altogether, and made every global symbol fluid instead. Fluids allow all the functionality of globals, plus they will allow other threads to locally bind the name.

Finally, I discovered an inherent design flaw in the Lisp language used by Reduce which I could not fix. A feature in Reduce allows the user to set flags to a symbol. These flags always affect the global symbol. Many Reduce packages use this feature, adding, retrieving and removing flags from shared global symbol names. I could not find a way to around this issue without modifying all of Reduce. This meant I was not able to parallelise the building of most Reduce packages.

Luckily, I did manage to find at least 20 packages which would not conflict in modifying symbol flags. I parallelised the building of those, the built the other packages sequentially. The resulting running time was `8.771s`. This showed a 40% improvement, which is very promising. I believe that fixing the issue of shared global names in Reduce could lead to a big improvement of build time when using multi-threading.

4.6 Testing ParVSL on different platforms

My benchmarks so far have focused on x86 Linux, running up-to-date compilers. VSL is a cross-platform language which should run on any system as long as the C++ compiler supports it. ParVSL further requires C++11 support, however at the time of this writing, the standard is already widely supported on all major platforms. I have also ran my test on different systems to verify this claim.

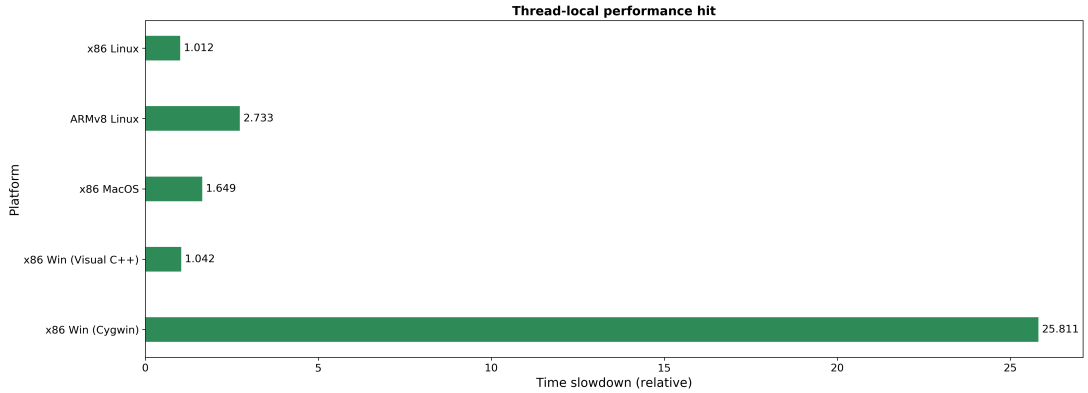
I tested three major operating systems: Linux, MacOS, and Windows. I also ran the tests on a Raspberry PI to test it on the ARM platform. Compiling on the Raspberry PI posed problems even for the single-threaded VSL, as only older compilers are available there and they exhibiting some bugs in compiling the code and complained about valid code. The other systems successfully compiled and ran all the code.

The more surprising result is the vast difference in performance between the platforms when running ParVSL vs VSL. The following are times

	VSL	ParVSL
x86 Linux	58s	1m08s
x86 MacOS	1m19s	3m56s
x86 Windows (Cygwin)	1ms	10m15

4.6.1 Thread-local access performance

The biggest culprit for the performance imact is the system's thread local storage (TLS) mechanism.



As we can observe, simply using thread-local storage can have a big impact on performance.

Chapter 5

Conclusion

I have successfully implemented a parallel programming language. The language ParVSL allows the user the use multi-threading to speed up their algorithms. It offers a simple shared memory model, based on mutual exclusion and condition variable, without compromising on any features of the original language VSL. I have demonstrated this by using it to build a large software project: the REDUCE Algebra System. I then used to implement a thread pool and tested it on parallel algorithms with inter-thread communication, proving a large performance gain can be obtained.

Ultimately, I have shown that a Computer Algebra System can benefit from parallelism, and have proved that it is possible to modify REDUCE, a large real-world application to use effectively employ multi-threading.

5.1 Cross platform performance

Historically, one of the biggest difficulty of implementing multi-threaded languages was providing cross platform support. I have tested ParVSL on the major platforms and showed that is capable of supporting them. However, I also discovered that performance across platforms is very inconsistent. System specific mechanisms of thread local storage and mutual exclusion have very different implementations and characteristics. This meant I could not replicate the performance achieved on Linux on other platforms. As of today, an understanding of each system and careful programming of individual scenarios is still necessary.

5.2 Parallelism is hindered by imperative programming

I discovered that the REDUCE Lisp language included a few historical design decisions which limit the potential of parallelism, namely its side-effectful nature. The functional programming paradigm makes it much easier to write safe high-performance multi-threaded by avoiding data races. When using an imperative style, the language has to do extra work to maintain safety, such as using mutual exclusion on variable access, which can severely slow down any algorithm. memory model which allows side-effects it is much more difficult

RLisp already offers many features for funtional programming. Modifying REDUCE to avoid side-effectful features of the language, and using it in a more functional would be needed for it to make good use of multi-threading.

5.3 Future work

While ParVSL was fully able to support all of the REDUCE, it is an interpreted language which limits its performance. A large number of optimisations such is unavailable in this case and many checks are performed real-time, rather than statically, slowing it down. The next step would be to use use the lessons learned to modify VSL's compiled brother: CSL. CSL is compiled and already much faster than VSL, however it has a much larger code base. While it would take more work, most of the lessons from writing ParVSL could be translated to ParCSL.

Bibliography

[1] Anthony C. Hearn and. *REDUCE Computer Algebra System*. 2009. URL: http://www.reduce-algebra.com/.

[2] C. J. Cheney. "A Nonrecursive List Compacting Algorithm". In: *Commun. ACM* 13.11 (Nov. 1970), pp. 677–678. ISSN: 0001-0782. DOI: 10.1145/362790.362798. URL: http://doi.acm.org/10.1145/362790.362798.

[3] Richard P. Gabriel and John McCarthy. "Queue-based Multi-processing LISP". In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. Austin, Texas, USA: ACM, 1984, pp. 25–44. ISBN: 0-89791-142-3. DOI: 10.1145/800055.802019. URL: http://doi.acm.org/10.1145/800055.802019.

[4] GNU Project. *GDB*. 1986. URL: https://www.gnu.org/software/gdb/.

[5] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. "Gprof: A Call Graph Execution Profiler". In: *SIGPLAN Not.* 17.6 (June 1982), pp. 120–126. ISSN: 0362-1340. DOI: 10.1145/872726.806987. URL: http://doi.acm.org/10.1145/872726.806987.

[6] Robert H. Halstead Jr. "Implementation of Multilisp: Lisp on a Multiprocessor". In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. Austin, Texas, USA: ACM, 1984, pp. 9–17. ISBN: 0-89791-142-3. DOI: 10.1145/800055.802017. URL: http://doi.acm.org/10.1145/800055.802017.

[7] Robert H. Halstead Jr. "MULTILISP: A Language for Concurrent Symbolic Computation". In: *ACM Trans. Program. Lang. Syst.* 7.4 (Oct. 1985), pp. 501–538. ISSN: 0164-0925. DOI: 10.1145/4472.4478. URL: http://doi.acm.org/10.1145/4472.4478.

[8] Valgrind Project. *Valgrind*. 2000. URL: http://valgrind.org/.

Appendix A

Source Code Listings

A.1 Cheney's algorithm

```
// LispObject is just a pointer type
typedef uintptr_t LispObject;

// LispObject is a pointer type
uintptr_t fringe1, limit1; // heap1, where all allocations happen
uintptr_t fringe2, limit2; // heap2 used for copying GC

LispObject allocate(size_t size) {
  if (fringe1 + size > limit1) {
    collect();
  }

  if (fringe1 + size > limit1) {
     // We are out of memory. Try to increase memory
     // ...
  }

  uintptr_t result = fringe1;
  fringe1 += size;
  return result;
}

// Two helper functions are needed
LispObject copy(LispObject obj) {
  size_t len = size(obj);

  LispObject new_obj = static_cast<LispObject>(fringe2);
  fringe2 += len;
  return new_obj;
}
```

```
uintptr_t copycontent(LispObject obj) {
  for (auto ref&: forward_references(obj)) {
    ref = copy(ref);
  }

  return static_cast<uintptr_t>(obj) + size(obj);
}

void collect() {
  // First we copy over the root set, which includes symbols.
  for (LispObject& symbol: symbols_table) {
    symbol = copy(symbol);
  }

  uintptr_t s = heap2;
  while (fringe2 < fringe2) {
    s = copycontent(static_cast<LispObject>(s));
  }

  swap(fringe1, fringe2);
  swap(limit1, limit2);
}
```

A.2 `Gc_guard` and `Gc_lock`

A.3 Lock free symbol lookup

A.4 Thread pool

```
  lisp;

  symbolic procedure queue;
      {nil, nil};

  symbolic procedure q_push(q, x);
      begin scalar back, newback;
          back := second q;
          newback := {x};

          if null back then <<
```

```
            rplaca(q, newback);
            rplaca(cdr q, newback); >>
        else <<
            rplacd(back, newback);
            rplaca(cdr q, newback); >>;
        return q;
    end;

symbolic procedure q_pop(q);
    begin scalar front, next;
        front := first q;
        if null front then
            return {}
        else <<
            next := cdr front;
            if null next then rplaca(cdr q, {});
            rplaca(q, next);
            return (first front); >>;
    end;

symbolic procedure q_empty(q);
    null (first q);

symbolic procedure atomic(val);
    {mutex(), val};

symbolic procedure atomic_set(a, val);
begin
    scalar m;
    m := first a;

    mutexlock m;
    rplaca(cdr a, val);
    mutexunlock m;
end;

symbolic procedure atomic_get(a);
begin
    scalar m, res;
    m := first a;

    mutexlock m;
    res := cadr a;
    mutexunlock m;
```

```
    return res;
end;

symbolic procedure safeq();
    {queue(), mutex(), condvar()};

symbolic procedure safeq_push(sq, x);
    begin scalar q, m, cv;
        q := first sq;
        m := second sq;
        cv := third sq;

        % print "safeq push getting mutex";
        mutexlock m;
        % print "safeq push got mutex";
        q_push(q, x);
        condvar_notify_one cv;
        % print "safeq push unlocking mutex";
        mutexunlock m;
        return sq;
    end;

symbolic procedure safeq_pop(sq);
    begin scalar q, m, cv, res;
        q := first sq;
        m := second sq;
        cv := third sq;
        res := nil;

        % print "safeq pop getting mutex";
        mutexlock m;
        % print "safeq pop got mutex";
        % print thread_id ();
        while q_empty q do condvar_wait(cv, m);
        res := q_pop q;
        % print "safeq pop unlocking mutex";
        mutexunlock m;
        % print "safeq pop done";
        return res;
    end;

% non-blocking call
symbolic procedure safeq_trypop(sq);
    begin scalar q, m, cv, res;
```

```
        q := first sq;
        m := second sq;
        cv := third sq;
        res := nil;

        % print "safeq trypop getting mutex";
        mutexlock m;
        % print "safeq trypop got mutex";

        if q_empty q then
            res := nil
        else
            res := {q_pop q};

        % print "safeq trypop unlocking mutex";
        mutexunlock m;
        return res;
    end;

symbolic procedure safeq_empty(sq);
    begin scalar r, m;
        m := second sq;
        % print "safeq empty getting mutex";
        mutexlock m;
        % print "safeq empty got mutex";
        r := q_empty (first sq);
        mutexunlock m;
        return r;
    end;

symbolic procedure future();
    {mutex (), nil};

% blocking call to wait for future result
symbolic procedure future_get(fut);
begin
    scalar m, state, cv, res;
    m := first fut;
    % print "future get getting mutex";
    mutexlock m;
    % print "future get got mutex";

    state := second fut;
```

```
    if state = 'done then <<
        res := third fut;
        mutexunlock m;
        return res >>;

    if state = 'waiting then
        cv := third fut
    else <<
        cv := condvar ();
        rplacd(fut, {'waiting, cv}) >>;

    % print "future waiting cv";
    condvar_wait(cv, m);
    % print "future got signaled cv";
    % ASSERT: promise is fulfilled here

    res := third fut;
    % print "future get unlocking mutex";
    mutexunlock m;

    return res;
end;

% non-blocking call for future result
% can wait on cv until timeout
symbolic procedure future_tryget(fut, timeout);
begin
    scalar m, state, cv, res;
    m := first fut;
    % print "future tryget getting mutex";
    mutexlock m;
    % print "future tryget got mutex";

    state := second fut;

    if state = 'done then
        res := {third fut}
    else if timeout = 0 then
        res := nil
    else <<
        if state = 'waiting then
            cv := third fut
        else <<
            cv := condvar ();
```

```
            rplacd(fut, {'waiting, cv}) >>;

        if condvar_wait_for(cv, m, timeout) then
            res := {third fut}
        else
            res := nil >>;

    % print "future tryget unlocking mutex";
    mutexunlock m;

    return res;
end;

symbolic procedure future_set(fut, value);
begin
    scalar m, state;
    m := first fut;

    % print "future set getting mutex";
    mutexlock m;
    % print "future set got mutex";
    state := second fut;

    if state = 'done then
        error("future already set");

    if state = 'waiting then
        condvar_notify_all third fut;

    rplacd(fut, {'done, value});

    % print "future set unlocking mutex";
    mutexunlock m;
end;

symbolic procedure tp_runjob(tp);
begin
    scalar tp_q, job, resfut, f, args, res;
    tp_q := first tp;
    job := safeq_trypop tp_q;
    if null job then thread_yield ()
    else <<
        job := first job;
        resfut := first job;
```

```
        f := second job;
        args := third job;
        res := errorset({'apply, mkquote f, mkquote args}, t);
        future_set(resfut, res);
        % print "done job"
    >>
end;

symbolic procedure thread_pool_job(tp_q, status);
begin
    scalar job, resfut, f, args, res, stat;
    % print "Started worker";
    job := safeq_trypop tp_q;
    repeat <<
        if job then <<
            % print "got job";
            job := first job;
            resfut := first job;
            f := second job;
            args := third job;
            % res := apply(f, args);
            res := errorset({'apply, mkquote f, mkquote args}, t);
            future_set(resfut, res);
            % print "done job";
        >> else <<
            % print "yielding";
            thread_yield ();
        >>;
        job := safeq_trypop tp_q;
        stat := atomic_get status;
    >> until (stat = 'kill) or (stat = 'stop and null job);
    % print "shutting down thread_pool worker";

    return nil
end;

symbolic procedure thread_pool(numthreads);
    begin scalar tp_q, status, threads;
        tp_q := safeq();
        status := atomic 'run;
        threads := {};
        % print "starting workers";
        for i := 1:numthreads do threads := thread2('thread_pool_job, {tp_q, status
        return {tp_q, status, threads};
```

```
    end;

symbolic procedure tp_addjob(tp, f, args);
begin
    scalar tp_q, status, resfut;
    tp_q := first tp;
    status := atomic_get (second tp);

    if not (status = 'run) then
        return nil
    else <<
        resfut := future ();
        % print "pushing job";
        safeq_push(tp_q, {resfut, f, args});
        return resfut;
    >>;
end;

symbolic procedure tp_stop(tp);
begin
    scalar threads;
    % print "tp_stop";
    atomic_set(second tp, 'stop);
    % print "atomic set";
    threads := third tp;
    for each td in threads do <<
        % print "joining thread"; print td;
        jointhread td;
        % print "joined thread";
    >>;
    % print "tp stopped";
    return nil;
end;

symbolic procedure tp_kill(tp);
begin
    scalar threads;
    atomic_set(second tp, 'kill);
    threads := third tp;
    for each td in threads do <<
        % print "joining thread"; print td;
        jointhread td;
        % print "joined thread";
    >>;
```

```
  end;

  % tp := thread_pool();

  end;
```

Appendix B

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Implementing Parallelism in Lisp for REDUCE

Andrei-Vlad Badelita, Trinity College
Originator: Dr Arthur C. Norman
11 October 2018

Project Supervisor: Dr Arthur C. Norman
Directors of Studies: Prof. Frank Stajano, Dr. Arthur Norman
Project Overseers: Prof. Jean Bacon, Prof. Ross Anderson, Dr. Amanda Prorok

Introduction

Computer Algebra System (CAS) programs provide utilities for manipulating mathematical expressions spanning many fields. They employ numerical algorithms to enable symbolic computations on objects such as polynomials or matrices.

Numerical algorithms have long been considered as good candidates for parallel algorithms. Oftentimes, the algorithms involve a large number of simple calculations or searching through many solutions.

Among the main currently available CAS applications, two are proprietary (Mathematica and Magma), while the three main open-source ones (Maxima, Axiom and Reduce) are all written on top of Lisp kernels which lack multi-threading support. This is most likely because they have all originally been written over thirty years ago, long before the appearance of multi-core personal computers.

Modern computers almost universally provide multiple processing cores, enabling parallelism. Moreover, while core counts are increasing, per-core performance improvements have slowed down.

The aim of this project is to prototype multi-threading support for REDUCE. The work would involve modifying a smaller Lisp implementation (VSL), which has been used

as a development playground before for techniques later introduced in the much larger kernel (CSL) that REDUCE normally uses. VSL is slower than CSL, but is more compact and manageable. It provides most of the features required to run all of REDUCE, while being fast enough to try interesting calculations.

I will augment this code in C++11, making it relatively easy to keep cross-platform compatibility. The current code will require modifications to ensure it is thread-safe, the garbage collector being a particularly interesting case. The language will use a shared memory model with mutexes and signals. I will attempt to make these modifications without adding a noticeable overhead to existing sequential code, using the current REDUCE tests as regression tests and benchmarks. Then I will rewrite a few of the numerical algorithms that are inherently parallelisable and perform further testing to assess the improvements in performance.

Starting point

The REDUCE Computer Algebra System is a long-standing open-source project, for which Dr. Arthur Norman is a maintainer. REDUCE runs on a LISP back-end, which I am going to modify.

VSL, the LISP language I am basing my work on, consists of around 4000 lines of C code. This code is not written with multi-threading support in mind. C++11 provides good support for multi-threading, which I will make heavy use of.

The REDUCE project comes with a suite of tests. I will use these as regression tests, however I will make modifications and add my own during evaluation to test the multi-threading component, and assess performance trade-offs.

Resources required

I will mainly use my personal laptop, which has a quad-core x86 CPU and 16GB of RAM, running Arch Linux. This will be enough to write the code, compile and evaluate. Eventually, I might test the compiler on different platforms(Windows, MacOS) and perhaps different architectures (e.g. RaspberryPi). I am able to provide all these resources myself.

I will use version control (Git) to manage both the project and the dissertation, uploading to an online mirror (GitHub). I will always keep both a local copy and a cloud-hosted one of the last version of my work. I will make regular backups on removable storage.

I will use the MCS service machine only to upload the project.

Work to be done

The project breaks down into the following:

1. Modifying the existing code to ensure thread-safety.

2. Changing the garbage collector to support safe allocation and collection on multiple threads,

3. Implementing the necessary threading primitives: create thread and wait for thread, atomics, mutexes, conditional variables, etc. These will be built on top of the their C++11 equivalents.

4. Reimplementing some REDUCE numerical algorithms to benefit from parallelism.

Success criteria

The project will be a success if the multi-threaded LISP language is functional, running both sequential and parallel code as expected.

To evaluate this, I will first test the storage allocation and garbage collection systems. This can be done reasonably early by writing a C++ framework to generate patterns of allocation and release of data. By calling internal functions directly, I can assess the validity and performance of the garbage collector.

When the entire compiler reaches a working state, I will start writing tests in Lisp. These tests will involve common examples of concurrency, such as a work queue used by multiple threads.

Finally, running REDUCE and passing the regression tests successfully is important. Eventually, I will attempt to extend REDUCE code to make use of multi-threading support. Vector and matrix operations are good candidates for this.

Possible extensions

Once the language is fully functional, I can continue to investigate REDUCE algorithms that benefit from the new features. There are two interesting algorithms which I can improve:

- Polynomial factorisation involves a combinatorial search stage at the end, which has the worst potential cost of the entire algorithm. It is a well-known bottleneck and could benefit from parallelisation.

- Groebner bases represent a case of critical-pair/completion precessing for working with ideals generated by multiple polynomial constraints. The worst-case cost can be double exponential. Research and other implementations in the field show that parallel search can give huge performance boosts.

Further optimisations of the language can be considered, both in terms of performance gains and also difficulty to implement. As an example, the current language is interpreted. Given enough time, it could be modified to support compilation, which would lead to better performance.

Timetable

I have split the timetable in ten work-packages, including eight fortnights and the two vacations. The longer vacation breaks will account for break time, and exam preparation, however they should still allow a minimum of three weeks of work on the project.

The workload is skewed towards the first few packets. This is meant to allow for extra flexibility in the latter part to deal with any particularly difficult aspects of the project.

Planned starting date is Thursday 18/10/2018.

1. **Michaelmas weeks 3–4(18/10/18 – 31/10/18):**
   DEADLINE 19/10/18: Submission of final Project Proposal(this document).
   Learn to use the LISP implementation. Understand the structure of the compiler code. Set up a working build with regression tests. Start identifying thread-unsafe code and modifying it.

2. **Michaelmas weeks 5–6(01/11/18 – 14/11/18):**
   Reimplement the Garbage Collector to support allocation from multiple threads. Ensure single-threaded behaviour is not affected.

3. **Michaelmas weeks 7–8(15/11/18 – 28/11/18):**
   Implement functions for multi-threading. Enable creation of threads. Write simple tests to show it works. Further work on garbage collector.

4. **Michaelmas vacation(29/11/18 – 16/01/19):**
   Implement primitives, such as mutexes and condition variables. Analyse the code to make sure it is thread safe. Have a working compiler.

5. **Lent weeks 1–2(17/01/19 – 30/01/19):**
   Run REDUCE on new compiler and analyse performance trade-off on sequential code. Work on progress report and presentation.

6. **Lent weeks 3–4(31/01/19 – 13/02/19):**
   DEADLINE 01/02/19: Submission of final Progress Report.
   07/02/19 – 12/02/19: Progress Report Presentations.

7. **Lent weeks 5–6(14/02/19 – 27/02/19):**
   Read about numerical algorithms used inside REDUCE and find good examples which benefit from parallelism.

8. **Lent weeks 7–8(28/02/19 – 13/03/19):**
   Implement multi-threaded versions of numerical algorithms and evaluate performance gain.

9. **Easter vacation(14/03/19 – 24/04/19):**
   Further evaluation of project. Final improvements to the compiler. Write the main chapters of the dissertation.

10. **Easter term 1–3 (25/04/19 – 15/05/19):**
    Further evaluation and completion of dissertation.

    DEADLINE 17/05/19: Submission of final Dissertation.