Andrei-Vlad Bădeliță

Implementing Parallelism in Lisp for REDUCE

Part II Computer Science Dissertation

Trinity College

May 2, 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

Date

Proforma

Candidate Number:	**2320D**
Project Title:	**Implementing Parallelism in Lisp for REDUCE**
Examination:	**Computer Science Tripos – Part II, 2019**
Word Count:	**11683**
Final code line count:	**3686**
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 system. 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 to replicate fully. Additionally, a number of tests and algorithms need to 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.

Acknowledgements

I would like to thank Dr Arthur Norman for all the invaluable help and support he offered during the making of this project.

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 Computer Algebra System (CAS). It enables symbolic manipulation of mathematical expressions and provides a wide range of algorithms to solve problems spanning many fields. 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-threading capabilities. My aim is to remove this limitation for REDUCE. The project is written in a language called RLisp, which is a specialised Lisp dialect with an Algol-like syntax.

1.2 VSL

There are multiple implementations of the Lisp backend REDUCE uses: PSL, CSL and VSL. Visual Standard Lisp (VSL) is an interpreted language written in `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. It exists to provide a test-bed for ideas that may later move to the much larger, compiled CSL version.

1.3 Benefits of multithreading

The idea of using parallel computing to speed up computer algebra has come up in research papers for many years [2], but much of the activity pre-dates the now ubiquitous multi-core CPUs used in modern computers and the amount of memory which they now provide. Moreover, advancements in single-core CPU performance have 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. Therefore, binding the performance of REDUCE to single-threaded performance limits the speed gains brought by new hardware. This project involves building the infrastructure REDUCE needs to take full advantage of today's hardware.

1.4 Code examples

To help explain the concepts that I introduce, code fragments will be used to show 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 Appendix A.

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 Lisp variant. I also used an additional language running on top of Lisp, called RLisp, in order to write code for REDUCE. When the syntax of the language gets in the way of readability, pseudo-code is used to aid the explanations.

1.5 Achievements

My project involved modifications to several thousand lines of C/C++ code, upgrading it to make better idiomatic use of C++ as well as adding concurrency support. It has also used raw Lisp code in bracket notation, along with the rebuilding of REDUCE, which starts off as Lisp, but is almost entirely written in its own language. REDUCE as a whole is around half a million lines of code and my system fully supports it. I have also coded demonstration programs to show how it can take advantage of modern multi-core processors to improve performance, given the thread support I now provide it with. The existing set of REDUCE test scripts have provided me with significant examples to demonstrate correctness and analyse the overheads of concurrency support on all the major platforms (Linux, Windows and MacOS), with some surprising differences between them.

Chapter 2

Preparation

2.1 ParVSL

I have forked the original VSL project into a new language which I named Parallel VSL, or ParVSL. ParVSL is fully backwards compatible with VSL and will be compared 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 into multiple files, fixing a few obvious bugs, and porting from C to C++.

2.2 From C to C++

The VSL language was written in the C programming language. 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 has to be provided by individual compilers and operating systems and can break between versions.

C++ is a superset of C and can compile existing, standard C code. 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), I do not have to think about these details and the code I write is fully portable. The only requirement is that a C++11 compliant compiler is used supporting the target platform. 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 mostly involved adding a few more

explicit casts[1]. However, I also transitioned the code to idiomatic C++ as I analysed more parts of it and became confident those changes wouldn't affect the semantics of the program.

2.2.1 RAII classes

The *RAII (Resource Allocation Is Initialisation)* design pattern is common in C++ [3, Item 37]. 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 (it goes out of scope or it is deleted), respectively.

We can use this mechanism to implement 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: for managing threads, shallow binding and for garbage collection synchronisation.

2.3 Throughput vs latency

When optimising for performance in a programming language, we have to analyse the trade-off 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.

In a CAS program the user is most likely to care about throughput, i.e. for computing the outputs 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 maximising throughput in the application. This is particularly important when designing the garbage collector.

2.4 The read-eval-print loop

The VSL Lisp system has a *read-eval-print* loop at its core: it reads the next instruction, evaluates it and prints the result. When evaluating performance, I carefully analysed the

[1] e.g. C++ is stricter than C about the need for `const` qualifiers on pointers that may refer to C-style strings.

functions called within this loop. These functions constitute the *critical path*. The code on the critical path is the most heavily used and should be as efficient as possible. When I modify it to enable multi-threading, I pay special attention to not encumber it.

2.5 Memory allocation

Memory is managed by the interpreter. VSL 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 the one in use 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 behave as if they were *joined* together to maintain the abstract model of contiguous memory. Binary search is used to identify the block containing a location.

The allocations are efficient, using a simple pointer, called *fringe* to indicate the start of the free area of memory. An additional pointer *limit* indicates the end of free memory. To allocate, VSL checks whether there is enough space, then simply increments the fringe by the specified amount. If the allocation would cross `limit`, it triggers *garbage collection*.

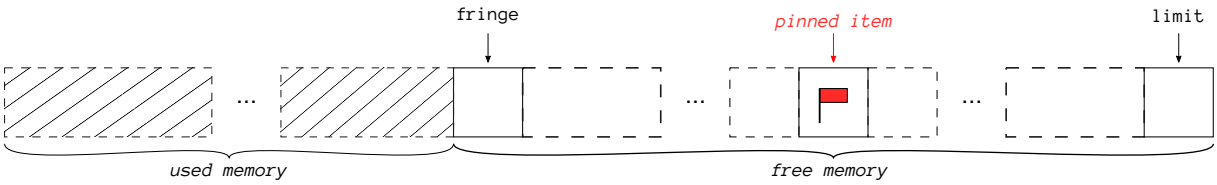

Figure 2.1: Memory allocations in VSL

2.6 Garbage collector

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

In effect, the garbage collector is an important component of VSL 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.

2.6.1 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*.

The algorithm is *stop-the-world*: it requires pausing execution of the program until the entire collection is over. The alternative, an *incremental* GC would use minor collections which only partially reclaim memory. Such an algorithm would only reduce latency at the expense of throughput and fragmentation.

Cheney's algorithm [4] is a way of implementing a 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 objects which are known to be in use. One example of elements in the root set is the set of symbols that are in use at the start of garbage collection. The stack also contains 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.

Using the root set, we can trace all references to build the reachable set. Objects may contain references to other objects, which are also considered reachable. All objects in the reachable set must be kept during collection, while everything else may be safely collected. Cheney's algorithm copies over the objects in the root set first. It also modifies their original location to become a *forwarding reference*: a note indicating that they were moved and what their new location is. Then, it scans the newly allocated memory (the objects copied over) for more references. For example, a list object will contain a reference to the rest of the list which is now also copied over.

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

2.6.2 Conservative GC

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. Another is to tag all data on the stack with an extra bit indicating which values are pointers. This approach is used in OCaml [5, Chapter 20], yet it still has the disadvantage of limiting integer types and requiring an extra instruction (i.e. shift) during runtime.

Instead, VSL uses a *conservative* GC, meaning it over-approximates the root set. 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 ordinary Lisp objects, which rules out copying them over. The solution was to *pin* them, i.e. mark them in the heap so they are not moved. Listing 1 shows the process of pinning objects. 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. When building the entirety of REDUCE,

the number of pinned items does not reach 300. Considering memory used is in the order of megabytes, these pinned locations cause negligible fragmentation.

```
int main() {
  ...
  // before starting the Lisp interpreter
  c_stack_base = approximate_stack_pointer();
  ...
}

void garbage_collection() {
  c_stack_head = approximate_stack_pointer();

  // scan the stack from its head to base
  for (uintptr_t s  = c_stack_head;
                 s  < c_stack_base;
                 s  += sizeof(LispObject))
    if (in_heap(s)) { // check if s points to the virtual heap
      set_pinned(s); // found an ambiguous root
    }
  ...
}
```

Listing 1: Scanning the stack before GC for ambiguous references.

2.6.3 Running single-threaded

I decided to run the garbage collector on a single thread. This approach still requires synchronisation with all running threads, however it requires no extra inter-locking during collection. Implementing a safe multi-threaded garbage collector efficiently would probably merit its own project. I believe that for the small hardware thread counts of current day computers a single threaded algorithm is adequate, not having to deal with data races and inter-locks. However, his may change as CPU core counts increase.

2.7 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* symbol 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.

2.8 Debugging tools

Debugging multi-threaded code is especially difficult and I needed the right tools to help me find the source of issues. I used the `gdb` [6] debugger to step through code and find the source of the problem. I used `gprof` [7] to profile my code and find the areas on the critical path which needed optimisations the most. Finally, I used `valgrind` [8] to analyse ParVSL for memory problems, data races and undefined behaviour.

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 called `thread` to VSL which takes a piece of code as an argument, starts a new thread and joins again with the main thread. Once implemented, I could run the 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. The interpreter is not thread-safe and data races on global variables (including printing to the same stream) lead to undefined behaviour. Although printing is possible, 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 or 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 areas of the interpreter.

To manage the threads, I made further use of the C++11 standard library. I devised 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. Subsequently, the user can then use the identifier to join the thread.

3.2 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. As such, I further split the memory

into regions, which I call *segments*. A segment is a thread-local region for allocation on the heap (see figure 3.1).

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*).

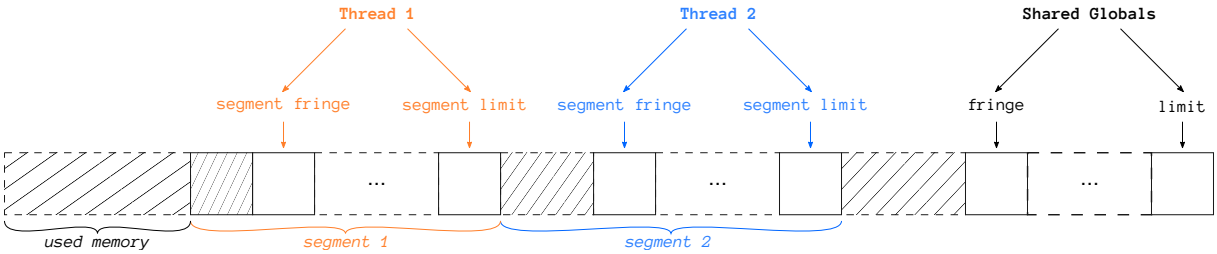

Figure 3.1: Memory allocations in ParVSL

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 segment 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 performed 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), it is allocated outside any segment, using the global fringe.

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 internal fragmentation and 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 the trade-off depends on the total memory, I have found a good compromise for segments of a few kilobytes each.

3.3 Garbage collection

The garbage collector 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.

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 didn't it would be possible for a thread to add references to the heap on its stack after those locations were marked safe to delete, causing corruption.

When a thread is initialised, I save its own stack base in `Thread_data`, and then also save its stack head when it is paused to wait for GC. All these stack ranges are scanned before I start garbage collection, as shown in 2.

```
void garbage_collection() {
  for (auto thread: thread_table)
    // scan the stack from its head to base
    for (uintptr_t s = thread.c_stack_head;
                   s < thread.c_stack_base;
                   s += sizeof(LispObject))
      if (in_heap(s)) { // check if s points to the virtual heap
        set_pinned(s); // found an ambiguous root
      }
  ...
}
```

Listing 2: Scan the stack of each thread for pinned items before GC.

3.3.1 Garbage collection locks and safety

The thread initiating garbage collection must wait for all threads to be ready. Similarly, to prevent starvation, all threads must regularly check whether a garbage collection cycle was initiated and make themselves safe.

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 they would need to call the GC. However, this solution is incomplete: 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. In figure 3.2, thread 2 for thread 1 to release but thread 1 was paused waiting for the GC, causing a deadlock. Similarly, any effectful computation, like waiting for user input will prevent the collection from starting.

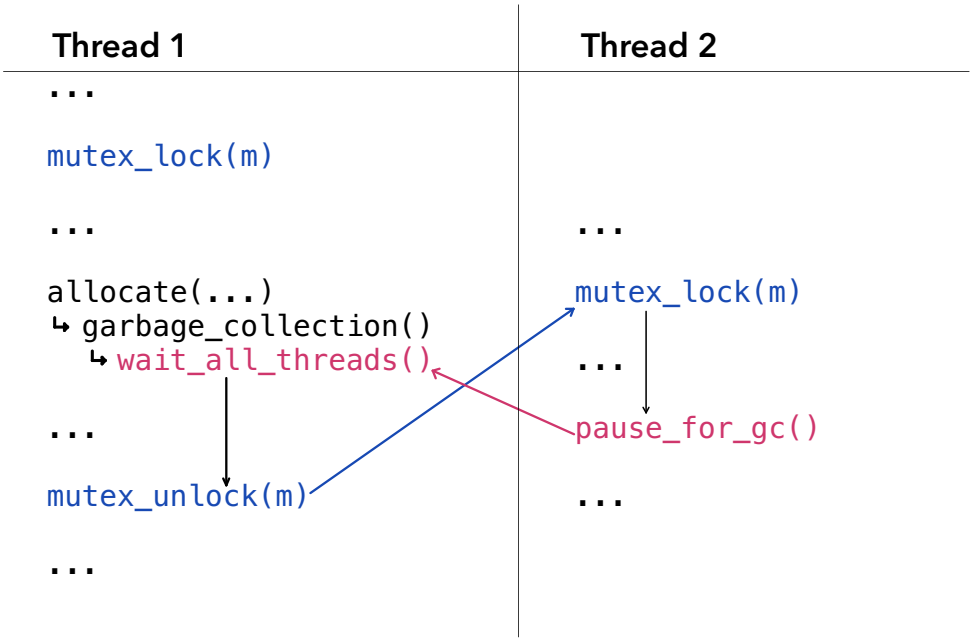

Figure 3.2: Possible deadlock when starting garbage collection

There are three scenarios in which a thread should check for GC. The first one is when it tries to allocate. Then, it should poll a global flag in the interpreter, so that time-consuming loops which do not allocate do not delay the garbage collector for too long. Finally, it must handle any blocking operation, especially when it depends on a signal from another thread (as in figure 3.2).

I created two classes to help me synchronise threads for GC and handle all the scenarios above: `Gc_guard` and `Gc_lock`. To begin with, I used the following shared global state:

```
std::atomic_int num_threads(0);

// number of threads ready for GC
std::atomic_int safe_threads(0);

// to wait for all threads to be GC-safe
std::condition_variable gc_wait_cv;

// to notify all threads that GC is completed
std::condition_variable gc_done_cv;

// flag indicating gc is running (or pending)
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. stack base and stack head) 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.

The `Gc_guard` class handles both scenarios. It has a constructor and a destructor and is a way for the thread to promise it is in a safe state.

```
class Gc_guard {
  // Global mutex shared by Gc_guard instance.
  static std::mutex gc_guard_mutex;

  Gc_guard() {
    td.c_stack_head = approximate_stack_pointer();

    // This thread is now safe for GC.
    safe_threads += 1;

    // Notify the thread waiting for garbage collection.
    gc_wait_cv.notify_one();
  }

  ~Gc_guard() {
    std::unique_lock<std::mutex> lock(gc_guard_mutex);

    // Wait until GC is done.
    gc_done_cv.wait(lock, []() { return !gc_on; });

    // Thread is not longer safe for GC.
    safe_threads -= 1;

    // Thread stack head will be invalidated when execution resumes.
    thread_data.c_stack_head = nullptr;
  }
};
```

Listing 3: `Gc_guard` makes a thread safe for GC.

The `Gc_guard` is designed to be used before any blocking call. For example, to fix the issue in figure 3.2, we simply need to modify the code in thread 2 as follows:

```
...
{
  Gc_guard gc_guard; // thread is now safe for GC
  mutex_lock(m); // this operation may block.

  // Gc_guard destructor will block here until GC is completed.
}
...
```

The `Gc_guard` class is accompanied by `Gc_lock`. A thread trying to initiate the GC has to acquire a `Gc_lock`. This lock makes further use of a mutex lock for mutual exclusion. Additionally, it waits until all threads are in a safe state.

```
class Gc_lock {
  // Global lock shared by Gc_lock instances.
  static std::mutex gc_lock_mutex;

  // Prevents other threads from acquiring the Gc_lock.
  std::unique_lock<std::mutex> lock;

  // The GC thread needs to be in a safe state as well.
  Gc_guard gc_guard;

  // Initialise the lock and guard before Gc_lock is constructed.
  Gc_lock() : lock(gc_lock_mutex), gc_guard() {
    gc_on = true;

    // Wait until all threads are in a safe state.
    gc_wait_cv.wait(lock, []() {
      return paused_threads == num_threads;
    });
  }

  ~Gc_lock() {
    gc_on = false;

    // Notify all threads that GC is over.
    gc_done_cv.notify_all();
  }
};
```

Listing 4: Garbage collection must be done under the `Gc_lock`.

3.4 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.

As before, the naïve 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 allow multiple threads to lookup symbols at the same time. However, as soon as one thread has to create a symbol, all the readers 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 did not find any. In the case of two threads looking up the same symbol, 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 serialises the writes, so it does prevent undefined behaviour in C++, however it may still create the same 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 a static array of linked lists. VSL never erases symbols, which made the lock-free implementation easier.

```
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_acquire_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 thread 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;
}
```

Listing 5: Lock-free symbol hashtable look up implementation

This approach adds no penalty to single-threaded code. For comparison, I also implemented the mutex lock approach and tested them on building REDUCE (single-threaded). Table 3.1 shows that the mutex version is noticeably slower, while the lock-free takes the same amount of time.

	Time
VSL	1m55s
ParVSL with mutex lookup	2m10s
ParVSL with lock-free lookup	1m55s
ParVSL with std::string	2m04s

Table 3.1: Symbol lookup effects on building REDUCE

3.4.1 Impact of C++ `std::string` on performance

While modifying this code, I saw the opportunity to change C-style string to C++ standard library ones. The lookup function, for example, actually takes a char array pointer and the string length as separate arguments. I even identified a bug in VSL where the wrong length was passed as a hard-coded number, and it prompted me to use the more modern `std::string` class. Surprisingly, this change alone slowed down ParVSL, as can be seen in the last row of Table 3.1. I made sure no more extra copying or construction of these strings was done than necessary. I reverted my change in this case, but I note it as an interesting example of the trade-off C++ features can bring.

3.5 Symbol access

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 the scope of `f`. Most dynamic languages, even weakly typed ones, like Javascript or Python, will consider the equivalent code valid, but will encounter a runtime error because `y` is not defined. Lisp, and VSL in particular, has a much 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 through 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
}
```

Listing 6: Implementation of `let` in VSL.

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 implement a form of deep binding with association lists storing local values, yet performance was a concern. While shallow binding has a constant factor, deep binding requires an associative data structure which would add non-negligible overheads to a critical area of the interpreter.

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. A globally unique array index is reserved for each local symbol. All threads reserve that location within their local arrays for the symbol.

```
thread_local vector locals;
vector shared_fluids;

LispObject par_value(LispObject symbol) {
  // [value] returns the symbol's global value
  LispObject val = value(symbol);

  if (is_global(symbol))
    return val; // global symbols remain unaffected

  int loc = to_int(val); // val is a location

  if (is_fluid(symbol) and locals[loc] == undefined)
    // No local value for this fluid. Return the global one.
    return shared_fluids[loc];

  // get the thread-local value at that location
  return locals[loc];
}
```

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

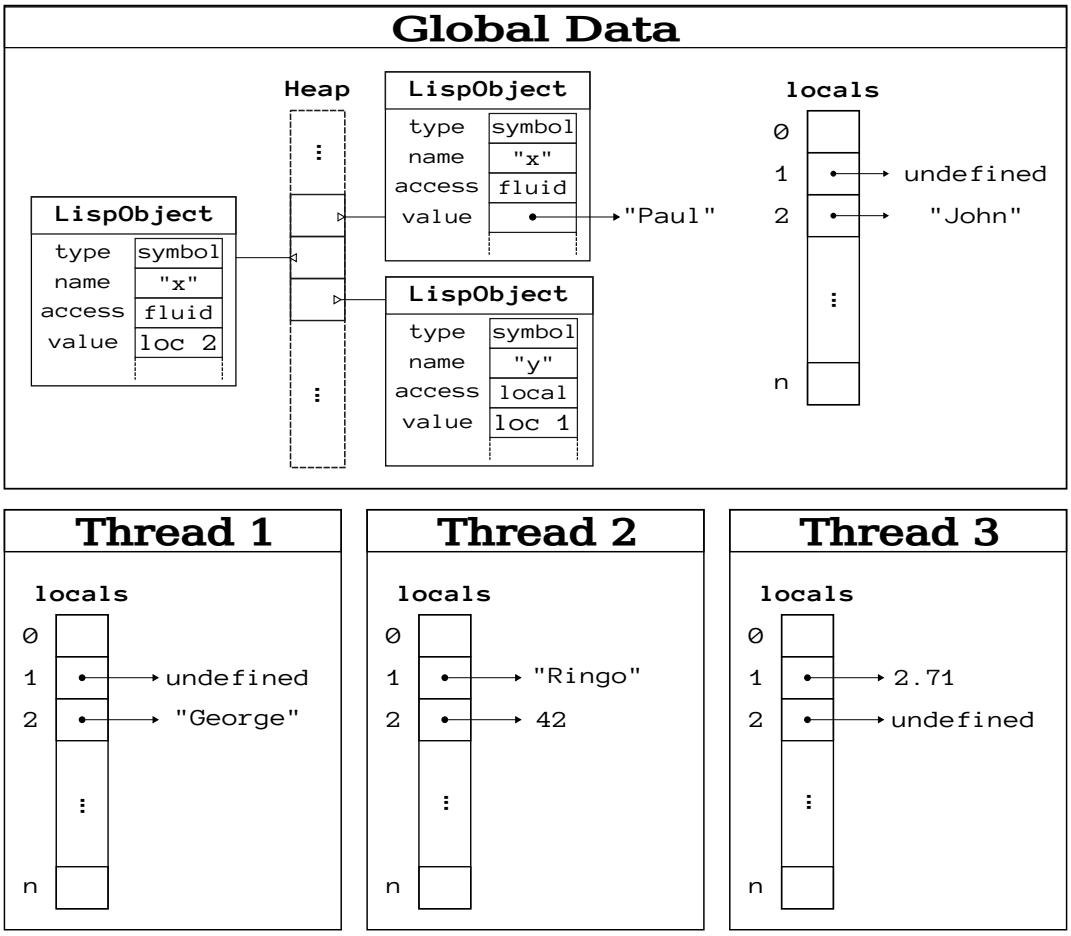

Figure 3.3: Storage of symbols in ParVSL

In figure 3.3 I show a full example of how symbols are stored. The first symbol is `g`, which is a *global* symbol. Its value is stored directly in the global heap (like in the original VSL). The symbol `x` is *fluid*. Its heap value simply indicates a location (`loc 2`). We use the index to find the actual value in thread-local storage. Threads 1 stores the string `"George"` while thread 2 stores the integer `42`. Thread 3 has not bound any local value to the symbol so it is `undefined`. However, it can still access the global value in `shared_fluids`, the string `"John"`. Finally, symbol `y` is *local*. It is undefined in `shared_fluids` but otherwise behaves the same as `x`, storing its thread-local values at index `1`.

I redesigned shallow binding to use a RAII style class, called `Shallow_bind` which deals with all the changes described above, while providing more safety guarantees, thus leading to more robust code. It checks if the symbol is global and prevents binding, it handles thread-local storage and it always restores the old value correctly in its destructor, including the case of exceptions. Previous code spent time testing error flags, manually restoring the old value in every place. For example, the code in listing 6 was rewritten as:

```
void implement_let(string varName, LispObject expr1, LispObject expr2) {
  LispObject symbol = lookup_symbol(varName);
  Shallow_bind bind_var{symbol, eval(expr1)};
```

```
  eval(expr2); // evaluate the rest of the code
}
```

Listing 7: Implementation of `let` in ParVSL.

3.6 Data races in the interpreter

Running VSL in parallel involves, in effect, running an interpreter in each thread. The spawned threads do not run a read-eval-print loop, instead they run a computation, save the result, then terminate. However, in ParVSL I allow these threads to execute any code the main thread can, with no restriction. This means all threads can use and modify global state and interact with standard input, output and files and thus can conflict with each other.

There are plenty of global variables holding state in VSL. I discovered which ones represented local state. One example is `unwind_flag`, which is heavily used for Lisp control flow. Among other uses, it is mutated to indicate an exception, to implement `goto` and `return` instructions, as well as to indicate early program exit, among other uses. Each thread needs its own version of this global flag, otherwise they will mess with each other's control flow, so I made it *thread-local*.

A third case is file handling. Global variables indicate which file is in use, and hold buffers for reading and writing. Again, I had to make all these thread-local. On the other hand, there are other variables holding truly global information, like file handles. At one point I was over-zealous in my modifications and made a variable thread-local which should have stayed global. Although it looked as if it indicated the direction of the current file, it turned out to be a bitmap of all the files in the system. The fact that VSL makes creative uses of integers for optimisation reasons effectively hides information from the type system. After a significant time spent debugging, I realised that the variable was used differently and the solution became apparent. I also modified the variable to use a proper *bitset* data structure, enhancing readability. As a bonus, ParVSL can now support a much larger number of open files than the 32 imposed by the size of integers.

Those globals which are not thread-local still pose the problem of data races. For instance, multiple threads may try to read from or write to the same file. ParVSL cannot avoid contention altogether: those conflicting operations will be serialised non-deterministically. However, it is important to prevent these data races at the C++ level, because they cause undefined behaviour, ultimately leading to anything from spurious crashes to more subtle data corruption.

I changed all functions for opening, closing, or altering a file to use mutex locks, with one mutex per file. Opening and closing files are not on the critical path of the interpreter so time was less of a concern. However, reading and writing can be part of the main loop. Many programs will run from a file, not standard input, and it is common to print regularly to standard output or file. To manage this, I added extra per-thread buffers to handle single character reads and writes. Then, I only synchronise threads when flushing the buffers, reducing contention. In my testing, these changes did not slow down ParVSL in any measurable way.

3.7 Threads and synchronisation in ParVSL

At the start of the project I bolted on to ParVSL a small, broken, but very much useful feature to evaluate Lisp code in a new thread, as demonstrated in section 3.1. I ran a number of small tests and found where this functionality broke the interpreter, then reworked those parts of the Lisp system until it successfully passed the tests. At the same time, I adapted a large suite of regression tests from REDUCE to ensure single-threaded behaviour was unaffected and performance didn't degrade significantly. Once I was confident that ParVSL could run multi-threaded I started implementing the extra functionality required to make parallel programs practical.

To start a thread, the user needs to call the `thread` function. Originally, the function only took a Lisp expression as argument and evaluated it. This was highly impractical, as the expression was passed unevaluated and data could not be passed naturally. The only way to allow communication was to write the data into a global symbol and have the code executed access that symbol.

I changed the implementation of `thread` to instead accept a function function name and the argument list as arguments. The arguments can be local symbols, and their values will be correctly bound to parameter names within the thread. The spawned thread then applies the function to the arguments and saves the result before terminating. The return value of `thread` is the thread id, which then has to be used to join the thread. Calling `join_thread` on the thread id, will block until the thread has finished executing, and retrieve the resulting value of the computation.

```
tid := thread('add, {2, 3});
result := thread_join tid;
print result;
> 5
```

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. Furthermore, all unjoined threads and corresponding return values need to be tracked. A hashmap from thread ids to return values is updated when thread execution is completed. Consequently, when `thread_join` is called this value is returned and removed from the hashmap. All threads have unique ids for the lifetime of the program, so there are never any conflicts in the hashtable. Starting and joining a thread are synchronised under a mutex lock to prevent data races.

I was careful to prevent the garbage collector from collecting these parameters and return values. Starting a thread is GC-safe: garbage collection will not initiate 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, so I add them to the unambiguous root set at the beginning of garbage collection.

3.7.1 Mutual exclusion and condition variables

It is not enough for the user to be able to spawn threads. ParVSL has a shared memory model so at a minimum it has to provide a way to mechanism for mutual exclusion. I exposed the C++ *mutex locks* with a very simple interface:

```
m := mutex();
mutex_lock(m);
mutex_unlock(m);
```

The mutexes are created in C++ and stored in a lock-free hashtable, each with a unique id. I used the lock-free hashtable implementation to allow different mutexes to be accessed in parallel. I also make use of the `Gc_guard` when `mutex_lock` is called. This is to ensure that the blocking call to lock the mutex does not cause a deadlock during garbage collection.

While ParVSL users can use mutexes to protect against data races, they may not always do so (due to omissions or bugs). Operations such as modifying a global symbol or mutating a list in place may still clash. One way to ensure safety would be to lock the mutex for any such operations, but this change would add a too much overhead to ParVSL. With this in mind, the decision was made to leave this issue as a responsibility of the ParVSL user, since they can manage the trade-off between safety and lock usage minimisation.

The second primitive that I added was *condition variables*. They provide a useful way for threads to signal each other, enabling an efficient way to perform common parallel processes such as waiting for a task to finish. I stored the condition variables the same way as the mutexes (i.e. in a lock-free hashtable) and the wait operation uses the `Gc_guard` to check for garbage collection.

I included a few more helpful function such as `hardware_threads` to check the number of hardware threads on the current system, `yield` to allow the system to reschedule the thread, and `sleep` to pause the thread for a specified duration.

These primitives are versatile enough to enable parallel algorithms, as I will showcase in Chapter 4. They can also be used to implement other common threading gadgets, such as atomics and futures (see 4.3.2 and 4.3.3).

3.8 Saving state to disk and reloading

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.

It is difficult to keep the same guarantees when multiple threads are running. If a thread attempts to preserve while others are running, the contents of the final image will be non-deterministic and generally undesirable, even if all data races were resolved. At the same time, I found little benefit to implementing such a feature, as the ability to

preserve is useful in retaining the results of computations, not intermediate states. Saving state to disk is used at the end of the program.

Once all threads are joined and there is only one thread running, all the thread-local data can be treated as global again and stored in the same locations that VSL uses. When restoring from an image, this data is moved back into thread-local storage. The image format is not affected.

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 RLISP code. Bracketed Lisp code was useful at the beginning of my project when ParVSL was not fully functional and I needed unit tests, while RLisp was necessary to write more complicated code which interfaces with REDUCE.

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 very large coverage test already available to demonstrate 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 detect development bugs and be confident that ParVSL retains backwards compatibility. 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 the source 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` [6] 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 discovered some 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 a system function to get the 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 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 shows 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 [7] tool to profile the code and found the biggest cause of the time difference is the symbol value extraction mechanism described in 3.5. 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

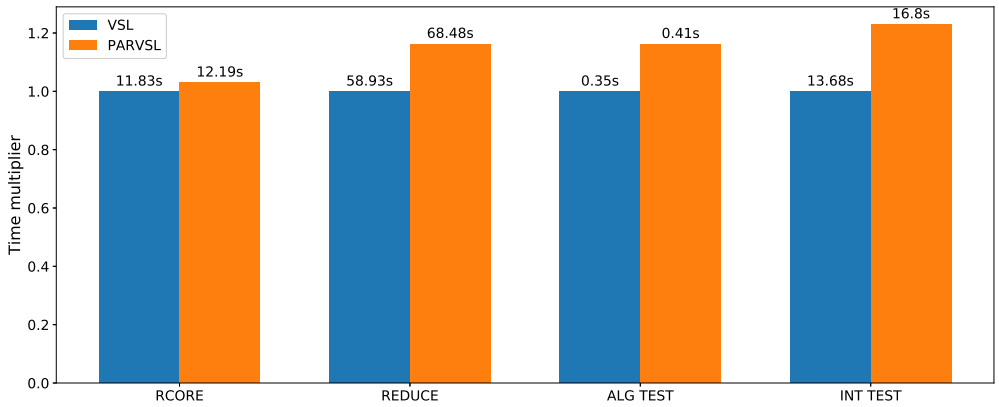

Figure 4.1: Relative performance of ParVSL on single-threaded tests (lower is better).

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 4.7.

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 once. The code below simply raises numbers to a large power in parallel. It helped me discover 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 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-threaded 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 be kept low so that the program never has to spawn more threads than the operating system limit, 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, but I used a FIFO 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. I wrote the following code in RLisp, since the thread pool will be useful in parallelising REDUCE code. 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 thread-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{q, m, cv});
begin
  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{q, m, cv}, x);
begin
  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[1], but workers can check the flag every time before taking a new task from the queue.

When the the user tries to stop an empty pool, all the workers will be in a sleeping state, waiting for the queue condition variable to be notified and 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 can also cause starvation as it delays other threads from being scheduled.

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

[1] C++ does not provide this as a primitive and indeed if a thread was interrupted in the middle of a system call or while holding a mutex lock it would cause all sorts of problems.

through global state, which would severely limit its usefulness. Passing arguments 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 Appendix A.7.

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 unwound 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. An efficient safe queue could be implemented using more 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, sorts 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 resulted in a new thread spawned for each element in the list. 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` can be called by any thread to run one job from 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 the `future_get` call 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

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 (as many as $O(N)$) 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). The following table shows the performance improvement we can gain by increasing the worker count.

	Number of workers			
List size ($\times$ 1000)	1	2	8	16
100	2.351	1.563	1.319	1.071
250	5.282	3.409	2.554	2.137
500	11.152	5.850	4.906	3.995
1000	22.840	11.706	9.054	7.514

Table 4.1: Parallel merge sort times by number of workers.

4.5 Multiplying polynomials in parallel

Polynomials are widely used in computer algebra. Being able to multiply two polynomials is a fundamental problem in computer algebra with for a plethora of applications ranging from including symbolic integration and computational geometry, so speeding it up with multi-threading could prove to be quite valuable.

At the same time, it is relatively easy to split the multiplication algorithm into multiple independent tasks. All the well-known multiplication algorithms (i.e. the naive quadratic solution, Karatsuba, and using the Fast Fourier Transform) have parallel versions [9, Chapter 30.3]. The latter are mostly used for multiplying numbers, and REDUCE uses the quadratic algorithm as it is better suited to sparse and more general multi-variate polynomials.

I wrote a parallel implementation of polynomial multiplication in REDUCE and ran it in ParVSL. It splits the polynomials into odd and even terms and thereby splits the problem into four sub-tasks (multiply every combination). These can be easily computed in separate threads, then joined to obtain the resulting polynomial. While the small test case only covers uni-variate polynomials the results I show might reasonably be expected to apply to more general cases.

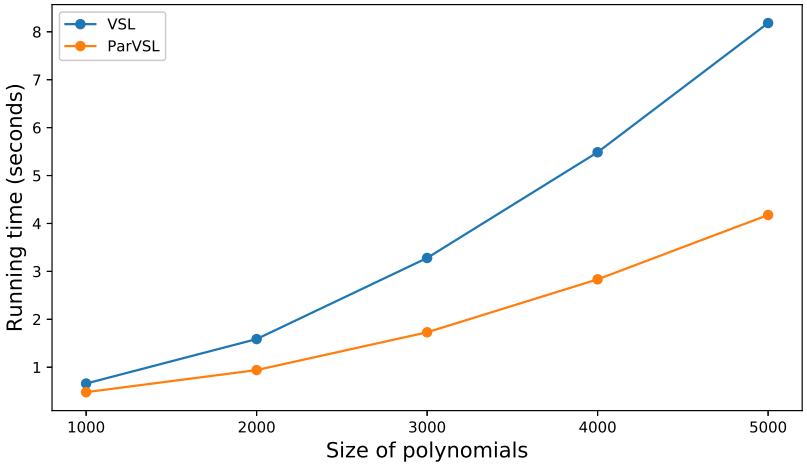

Figure 4.2: Running time of polynomial multiplication in parallel vs single-threaded (lower is better).

4.6 Parallel building of REDUCE

Most of REDUCE is made up of RLISP code, which can be run in VSL or ParVSL to generate a build. 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

A 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 it into 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 14.1 seconds. 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. Afterwards, I wrote a Python script which builds each package individually with the build flag on and saved those accessed to a file. The script computed all conflicts between packages, 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 instead access locally bound ones. This removes all conflicts between symbol accesses, without needing to modify code in the REDUCE packages themselves. It also illustrates how in the future REDUCE will need modification to cope with concurrency.

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 solved 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 8.7 seconds (vs 14.1s), a 40% improvement. I believe that fixing the issue of shared global names in REDUCE could lead to a significant improvement of build time when using multi-threading. The speed-up in system rebuild time will be valuable for developers.

4.7 Testing ParVSL on different platforms

My benchmarks so far 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 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. Table 4.2 shows the results.

	VSL	ParVSL
x86 Linux	58s	1m08s
x86 MacOS	1m19s	3m56s
x86 Windows (Cygwin)	1m03s	10m15s

Table 4.2: Building REDUCE on different platforms in VSL vs ParVSL.

4.7.1 Thread-local access performance

Once observed, this can be verified by trying a trivial test program just referencing a thread-local variable in a loop. Figure 4.3 shows a huge unexpected extra cost under Cygwin, and noticeable slowdown on the other platforms.

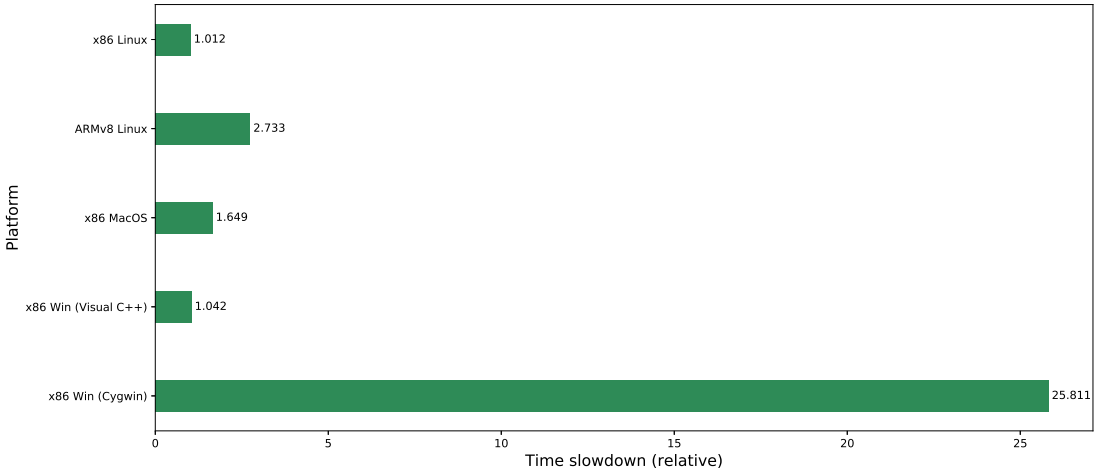

Figure 4.3: Thread local performance on different platforms (lower is better).

The biggest culprit for the performance impact is the system's thread local storage (TLS) mechanism. In the best case (i.e. on x86 Linux) a thread-local variable is accessed similarly to a regular static one. One would hope that Cygwin developers will implement a more efficient TLS in the future. Currently, the only two platforms with a negligible TLS slowdown are Linux and (native) Windows.

Chapter 5

Conclusion

I have successfully implemented a parallel programming language. The language ParVSL allows the user to employ multi-threading to speed up their algorithms. It offers a simple shared memory model, based on mutual exclusion and condition variables, 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 it 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 employ multi-threading effectively. Developers of REDUCE are now able to use ParVSL to test parallel versions of their algorithms, and some of these developers have already started testing my system on their workflows.

5.1 Cross platform performance

Historically, one of the biggest difficulties 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 programs 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 functional programming. Modifying REDUCE to avoid side-effectful features of the language, and using a more functional approach would be needed for it to make good use of multi-threading.

5.3 Future work

While ParVSL is able to support all of REDUCE, it is an interpreted language which limits its performance. A large number of optimisations are 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 brother: CSL. CSL is compiled and 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 create a similar ParCSL.

While I have focused on the implementation of the Lisp system, I did not formally define the semantics the new features. I talked about the trade-offs between safety and performance, and these need to be carefully considered when establishing the set of guarantees to provide for multi-threaded programs.

Bibliography

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

[2] Jean Della Dora. *Computer Algebra and Parallelism*. Ed. by John Fitch. Orlando, FL, USA: Academic Press, Inc., 1989. ISBN: 012209042X.

[3] Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. 1st. O'Reilly Media, Inc., 2014. ISBN: 1491903996, 9781491903995.

[4] 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.

[5] Yaron Minsky. *Real World Ocaml: Functional programming for the masses*. O'Reilly Media, Nov. 2013. ISBN: 144932391X.

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

[7] 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.

[8] Valgrind Documentation Team. *Valgrind 3.11 Reference Manual*. United Kingdom: Samurai Media Limited, 2015. ISBN: 9789888381210, 9888381210.

[9] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

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) {
  // skip over pinned items
  if (is_pinned(fringe1)) {
    fringe1 += sizeof(LispObject);
  }

  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);

  if (is_pinned(fringe2)) {
    // skip over pinned items;
    fringe2 += sizeof(LispObject);
  }

  fringe2 += len;
  return new_obj;
}

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

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

void collect() {
  c_stack_head = approximate_stack_pointer();

  // scan the stack from its head to base
  for (uintptr_t s  = c_stack_head;
                 s  < c_stack_base;
                 s  += sizeof(LispObject))
  if (in_heap(s)) { // check if s points to the virtual heap
    set_pinned(s); // found an ambiguous root
  }

  // remember the starting point of copied locations
  uintptr_t start_fringe2 = fringe2;

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

  uintptr_t s = start_fringe2;
  while (s < fringe2);
    s = copycontent(static_cast<LispObject>(s));
  }
```

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

A.2 Gc_guard and Gc_lock

```
namespace {
std::atomic_int num_threads(0);
std::atomic_int paused_threads(0);
std::condition_variable gc_waitall;
std::condition_variable gc_cv;
std::atomic_bool gc_on(false);

std::mutex gc_guard_mutex;
std::mutex gc_lock_mutex;
}

class Gc_guard {
private:
  // Global mutex shared by Gc_guard instance.
  static std::mutex gc_guard_mutex;

public:
  Gc_guard() {
    td.c_stack_head = approximate_stack_pointer();

    // This thread is now safe for GC.
    safe_threads += 1;

    // Notify the thread waiting for garbage collection.
    gc_wait_cv.notify_one();
  }

  ~Gc_guard() {
    std::unique_lock<std::mutex> lock(gc_guard_mutex);

    // Wait until GC is done.
    gc_done_cv.wait(lock, []() { return !gc_on; });

    // Thread is not longer safe for GC.
    safe_threads -= 1;
```

```
    // Thread stack head will be invalidated when execution resumes.
    thread_data.c_stack_head = nullptr;
  }
};

class Gc_lock {
private:
  // Global lock shared by Gc_lock instances.
  static std::mutex gc_lock_mutex;

  // Prevents other threads from acquiring the Gc_lock.
  std::unique_lock<std::mutex> lock;

  // The GC thread needs to be in a safe state as well.
  Gc_guard gc_guard;

public:
  // Initialise the lock and guard before Gc_lock is constructed.
  Gc_lock() : lock(gc_lock_mutex), gc_guard() {
    gc_on = true;

    // Wait until all threads are in a safe state.
    gc_wait_cv.wait(lock, []() {
      return paused_threads == num_threads;
    });
  }

  ~Gc_lock() {
    gc_on = false;

    // Notify all threads that GC is over.
    gc_done_cv.notify_all();
  }
};
```

A.3 Thread-local data

```
class Thread_data {
public:
    static constexpr int SEGMENT_SIZE = 0x10000; // 64KB per segment
    // use these as the segment we can write on.
    uintptr_t segment_fringe = -1;
    uintptr_t segment_limit = 0;
```

```
    int id; // thread id
    LispObject *c_stack_base;
    LispObject *c_stack_head = nullptr;

    LispObject work1 = NULLATOM;
    LispObject work2 = NULLATOM;
    LispObject cursym = NULLATOM;

    char boffo[BOFFO_SIZE+4];
    size_t boffop;

    int lispin = STDIN, lispout = STDOUT;
    std::string file_buffer[MAX_LISPFILES];

    char input_line[INPUT_LINE_SIZE];
    size_t input_ptr = 0, input_max = 0;

    char printbuffer[32];

    int curchar = '\n', symtype = 0;

    unsigned int unwindflag = unwindNONE;
    int backtraceflag = -1;

    // I suspect that linelength and linepos need to be maintained
    // independently for each output stream. At present that is not
    // done. And also blank_pending.
    int linelength = 80, linepos = 0, printflags = printESCAPES;
    bool blank_pending = false;

    /**
     * Whether the thread is in a safe state for GC.
     * */
    bool safe_memory = true;
};

thread_local Thread_data thread_data;
```

A.4 Shallow binding

```
class Shallow_bind {
private:
  int loc;
```

```
  LispObject save;
public:
  Shallow_bind(LispObject x, LispObject tval) {
    if (is_global(x)) {
      error1("shallow bind global", qpname(x));
    }

    loc = qfixnum(qvalue(x));
    LispObject& sv = td.local_symbol(loc);
    save = sv;
    sv = tval;
  }

  Shallow_bind(Shallow_bind&&) noexcept = default;

  ~Shallow_bind() {
    td.local_symbol(loc) = save;
  }
};
```

A.5 Lock free symbol lookup

```
/**
* [search_bucket] searches a particular bucket in the symbol table
* It can search the whole bucket, or down to a location.
* If specifying [stop] make sure it is in the bucket, otherwise it will
↪  loop.
* Returns -1 if not found.
* */
LispObject search_bucket(LispObject bucket, const char *name, size_t len,
↪  LispObject stop=tagFIXNUM) {
    for (LispObject w = bucket; w != stop; w = qcdr(w)) {
        LispObject a = qcar(w);    // Will be a symbol.
        LispObject n = qpname(a);      // Will be a string.
        size_t l = veclength(qheader(n)); // Length of the name.

        if (l == len && strncmp(name, qstring(n), len) == 0) {
            return a;                  // Existing symbol found.
        }
    }

    return -1;
}
```

```
LispObject lookup(const char *name, size_t len, int flag)
{
    size_t loc = 1;
    for (size_t i = 0; i < len; i += 1) loc = 13 * loc + name[i];
    loc = loc % OBHASH_SIZE;

    LispObject bucket = obhash[loc].load(std::memory_order_acquire);
    LispObject s = search_bucket(bucket, name, len);

    if (s != -1) return s; // found the symbol

    if ((flag & 1) == 0) return undefined;
    LispObject pn = makestring(name, len);
    LispObject sym = allocatesymbol(pn);

    LispObject new_bucket = cons(sym, bucket);

    while (!obhash[loc].compare_exchange_strong(bucket, new_bucket,
    ↪  std::memory_order_acq_rel)) {
        LispObject old_bucket = bucket;
        bucket = obhash[loc].load(std::memory_order_acquire);

        // Reaching here means bucket is constructed from old_bucket
        s = search_bucket(bucket, name, old_bucket);

        if (s != -1) return s; // another thread has created the symbol
        ↪  in the meantime

        new_bucket = cons(sym, bucket);
    }

    // successfully inserted the symbol in the hash, can return it.
    return sym;
}
```

A.6 Thread-local symbols

```
std::atomic_int num_symbols(0);

thread_local std::vector<LispObject> fluid_locals;
std::vector<LispObject> fluid_globals; // the global values
```

```
/**
 * Returns a shared id to index the symbol.
 * Used internally to identify the same symbol name on multiple
 * threads. This location will be an index into the actual storage array,
 * whether it's global or thread_local, and will be stored inside
 * qvalue(x).
 * */
int allocate_symbol() {
  std::unique_lock<std::mutex> lock(alloc_symbol_mutex);
  int loc = num_symbols;
  num_symbols += 1;
  fluid_globals.resize(num_symbols, undefined);
  return loc;
}

/**
* [local_symbol] gets the thread local symbol. may return undefined.
* Note, thread_local symbols are only lazily resised. Accessing a local
* symbol directly is dangerous. You need to use this function to ensure
↪  the
* local symbol is at least allocated.
*/
LispObject& local_symbol(int loc) {
  if (num_symbols > (int)fluid_locals.size()) {
    fluid_locals.resize(num_symbols, undefined);
  }

  return fluid_locals[loc];
}

/**
* [symval] returns the real current value of the symbol on the current
↪  thread.
* It handles, globals, fluid globals, fluid locals and locals.
* should be used to get the true symbol, instead of qvalue(s).
*/
LispObject& par_value(LispObject s) {
  if (is_global(s)) {
    return qvalue(s);
  }

  int loc = qfixnum(qvalue(s));
  LispObject& res = td.local_symbol(loc);
```

```
  // Here I assume undefined is a sort of "reserved value", meaning it
  ↪  can only exist
  // when the object is not shallow_bound. This helps me distinguish
  ↪  between fluids that
  // are actually global and those that have been bound.
  // When the local value is undefined, I refer to the global value.
  if (is_fluid(s) && res == undefined) {
    return fluid_globals[loc];
  }
  // This is either local or locally bound fluid
  return res;
}
```

A.7 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;

  mutexlock m;
  q_push(q, x);
  condvar_notify_one cv;
  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;

  mutexlock m;
  while q_empty q do condvar_wait(cv, m);
  res := q_pop q;
  mutexunlock m;
  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;

  mutexlock m;

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

  mutexunlock m;
  return res;
end;

symbolic procedure safeq_empty(sq);
begin scalar r, m;
  m := second sq;
  mutexlock m;
  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;
  mutexlock m;

  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}) >>;


  condvar_wait(cv, m);
  % ASSERT: promise is fulfilled here

  res := third fut;
  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;
  mutexlock m;

  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 >>;

  mutexunlock m;

  return res;
end;

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

  mutexlock m;
  state := second fut;

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

  if state = 'waiting then
    condvar_notify_all third fut;

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

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

  return nil
end;

symbolic procedure thread_pool(numthreads);
begin scalar tp_q, status, threads;
  tp_q := safeq();
  status := atomic 'run;
  threads := {};
  for i := 1:numthreads do
    threads := thread2('thread_pool_job, {tp_q, status}) . threads;
  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 ();
    safeq_push(tp_q, {resfut, f, args});
    return resfut;
  >>;
end;

symbolic procedure tp_stop(tp);
begin
  scalar threads;
  atomic_set(second tp, 'stop);
  threads := third tp;
  for each td in threads do <<
    jointhread td;
  >>;
  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 <<
    jointhread td;
  >>;
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.