Computer Science Tripos – Part II – Project Proposal

Implementing Parallelism in Lisp for REDUCE

Andrei-Vlad Badelita, Trinity College

Originator: Dr Arthur C. Norman

15 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. Setup 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.
   Work

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.