Work Package 5.2: A Report

Alasdair Macindoe

Acknowledgements

We acknowledge financial support from the OpenDreamKit Horizon 2020 European Research Infrastructures project (#676541). Further we would like to thank Dr Markus Pfeiffer for his supervision and all his help that was invaluable for this project.

Introduction

Whilst enumerating finite semigroups is a computationally expensive there are algothims that can run - in a practical sense - faster than naively enumerating the semigroup. One example of this algorithm was published in 1997 by Froidure and Pin[2] (which will hereby be called the Froidure-Pin algorithm). Unfortunately this algorithm does not run concurrently which is a problem for modern computational problems. This leads to another version of Froidure-Pin to be published[4] (which I will henceforth call the concurrent Froidure-Pin) for which a C++ implementation exists[1].

The aim of this project was to re-implement the concurrent Froidure-Pin algorithm in HPC-GAP[3] and analyse its scalability. The full repository for this project can be found on Github[5].

Structure

All the code can be located on Github[5]. The following may be of interest:

`README.md`: Installation and testing instructions

`versions.md`: Explanation of implementation details

`runtimes.md`: Raw data from experiments

`runtimes.md`: Explanation of different semigroups for testing

`experiment.g`: File to run experiments

`gap/`: Various implementations

Algorithm

The following is a computational explanation of the algorithm (without implementation details) for those who are more computationally minded than mathematically.

Data Structures

The following data structures are required:

1. `A fragment data structure`: A fragment consists of a value `K` and a list of `reduced words`
2. `A queue data structure`: In order to remain lockless each job cannot communicate with any other. This means there is no way, in ApplyGenerators, for any specific job to know if another job has the same word already. This queue data structure will hold any word that a job beleives *may* be new
3. `A word data structure`: To hold any and all relevant information on a specific word

Additionally the following methods are required:

1. An entry point
2. ApplyGenerators
3. ProcessQueues
4. DevelopLeft
5. A method for deterministically determining the bucket for a word

Further a way to obtain the following information (either through a data structure or through method) are required:

1. A word's suffix
2. A word's prefix
3. A word's right multiplications already calculated
4. A word's left multiplications already calculated
5. A word's first letter
6. A word's last letter

Additional methods or information may be required on how you choose to implement the algorithm.

Bibliography

[1] J.D Mitchell et al. Semigroups - gap package. *https://github.com/gap-packages/Semigroups*.

[2] Véronique Froidure and Jean-Eric Pin. *Algorithms for computing finite semigroups*, pages 112–126. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[3] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.8.8*, 2017.

[4] J. Jonušas, J. D. Mitchell, and M. Pfeiffer. Two variants of the Froiduire-Pin Algorithm for finite semigroups. *ArXiv e-prints*, April 2017.

[5] Alasdair G. Macindoe. Parallel semigroups gap. *https://github.com/Alasdair-Macindoe/parallel-sgrp-gap*.