# 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 computationally expensive there are algorithms 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[3] (which will hereby be called the Froidure-Pin algorithm). Unfortunately this algorithm does not run concurrently which is a problem for modern computational problems; thus this lead to another version of Froidure-Pin to be published[5] (which we will henceforth call the concurrent Froidure-Pin) which can run concurrently and for which a C++ implementation exists[1].

### Aim

The aim of this project was to re-implement the concurrent Froidure-Pin algorithm in HPC-GAP[4] and analyse its scalability. The full repository for this project can be found on Github[6]. This code was then shared to the general public under the GNY GPL v3 license.

## Structure

All the code can be located on Github[6]. The following may be of interest:

`README.md`: Installation and testing instructions

`versions.md`: Explanation of implementation details

`runtimes.md`: Raw data from experiments

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

`experiment.g`: File to run experiments

`gap/`: Various implementations

`tst/`: The test file

## Failed Attempts and Approach

This was a very challenging project, and there are many attempts before we arrived at `Version 2.1` which were failures. The first attempt was creating in GAP the simplistic `naiveenumeration` variation, then from there work on implementing the Froidure-Pin algorithm itself.
The real difficulty began with the concurrent Froidure-Pin algorthim which was written in some very dense mathematical notation. We have an attempt `concurrentfp` where we completely misunderstood the algorithm, we have `Version 1.x` which whilst do perform the job correctly and not truly concurrent at every stage and require locking. The insights gained from these failures did allow us to produce `Version 2.1` which we are incredibly happy with.
Overall we feel like this approach was the most beneficial. Not only did it require learning about semigroups and GAP, but it allowed us to start from the beginning and develop our way through the approaches done throughout the years and improve and learn ourselves from our failures.

## Implementation

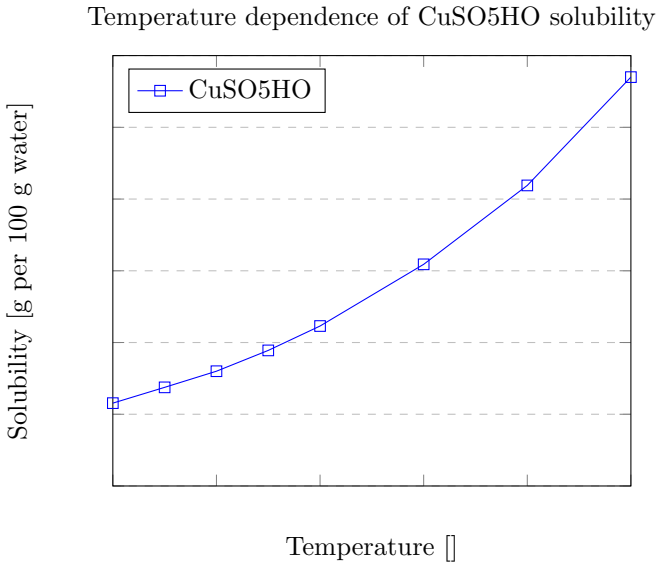
`Version 2.1` is an exact implementation of the version given in the paper. Since the paper does not dictate what data structures should be used the full implementation details can be found in `versions.md`. This gives very detailed overview of the algorithm and why it can run locklessly, as well as the reasons for the specific data structures.

## Analysis

Please refer to `runtimes.md` for the raw data, only summaries are provided below.
All experiments are on implementation `Version 2.x`.

### Experiment 1

The aim of this experiment was to see the effect of increasing the number of core available without increasing the number of fragments.
We believe this was worth investigating because HPC-GAP's Task implementation is not guaranteed 1 to 1 with processes.




## Contributions to HPC-GAP

A bug in HPC-GAP was discovered which would cause a segmentation fault when an empty list was attempted to be migrated between tasks. This was fixed by Dr Markus Pfeiffer[7].

## Future Work

We have identified some potential future works that could be done:

1. Fix any outstanding bugs and perform more significant testing
2. Covert data structures across to the HPC-GAP's datastructures package[2]
3. Investigate performance increases by using HPC-GAP's threads and processes API
4. Investigate performance increases in HPC-GAP's tasks system

## Conclusion

In sum we have successfully implemented the concurrent Froidure-Pin algorithm several times in HPC-GAP and compared their scalability. Additionally we have identified bugs that exist within HPC-GAP and identified future work that could be conducted in this area.
We further believe that the skills honed here will allow us to make stronger

Computer Scientists and software engineers. [[TO BE COMPLETED WHEN WE HAVE RESULTS]]

# Bibliography


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

[2] Max Horn et al. Datastructures. *https://github.com/gap-packages/datastructures*.

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

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

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

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

[7] Markus Pfeiffer. Fix migrateobjects for empty list of objects. *https://github.com/gap-system/gap/pull/1602*.