

Faculty of Science, Technology and Arts

**Department of Computing  
Project (Technical Computing)  
[55-604708]  
2019/20**

<b>Author:</b>	Liam Warner
<b>Student ID:</b>	26018527
<b>Year Submitted:</b>	2020
<b>Supervisor:</b>	Ho, Hsi-Ming
<b>Second Marker:</b>	Cooper, Martin
<b>Degree Course:</b>	BSc Computer Science (Games)
<b>Title of Project:</b>	A technical evaluation of Nim's language features and concepts

**Confidentiality Required?**

**NO** ☒

**YES**

# Abstract

This paper's purpose is to provide a technical evaluation of the language features and efficiency of the Nim programming language. This is done by a series of small test programs written in both Nim and a language that would best compete against it for the purpose of the program.

These programs are written to do the same abstract work and provide the same output, but language features and efficiencies are used when the language or toolchain provides them. Data is then collected when appropriate and the 2 programs compared.

This paper concludes that while Nim does provide a computational boost due to its compilation to C/++, it lags behind its competitors by its overall lack of compelling language features, young library ecosystem, and its occasional unreliable compiler errors.

# Contents

1	Introduction & Overview .....	1
1.1	Project Overview.....	1
1.2	The Nim Programming Language.....	1
1.3	Aims & Objectives .....	1
2	Research.....	2
2.1	Garbage Collection.....	2
2.1.1	Nim's GC Modes.....	3
2.1.2	Valgrind's Massif .....	3
2.2	Multithreading .....	4
2.2.1	Pthread & Managed Threads.....	4
2.2.2	C# Threading Abstractions .....	4
2.2.3	Task Based Multithreading .....	4
2.2.4	PLINQ.....	4
2.3	Data Gathering.....	5
2.4	JavaScript Compilation.....	6
2.4.1	Front End.....	6
2.4.2	Node JS.....	6
3	Design.....	6
3.1	Planning & Management .....	6
3.2	Applications.....	8
3.2.1	Boids.....	8
3.2.2	Prime Factors .....	8
3.2.3	Web Mail Client.....	8
4	Implementation .....	10
4.1	Boids.....	10
4.1.1	Code .....	10
4.1.2	Data Measuring.....	13
4.1.3	Findings .....	13
4.1.4	Testing.....	14
4.2	Multithreading .....	14
4.2.1	Code .....	14
4.2.2	Findings .....	18
4.3	Web Client.....	18
4.3.1	Code .....	18
4.3.2	Findings .....	20
5	Evaluation .....	21

5.1	Data Recording.....	21
5.2	Organization.....	21
5.3	Nim.....	21
5.4	Incomplete Projects .....	22
5.5	Personal Development.....	22
5.6	Conclusion.....	23
6	Appendices.....	24
6.1	A - Boids .....	24
6.2	B – Boid Data.....	24
6.2.1	Overview of GC modes.....	24
6.2.2	Comparing Nim with and without GC .....	24
6.2.3	Comparison of time taken to add Boids to the sequence .....	25
6.2.4	Comparison of time taken to calculate Boid movements.....	25
6.2.5	Snapshots of GC usage during execution.....	25
6.2.6	Comparison of Occupied Heap Memory.....	25
6.2.7	Comparison of complete program times .....	25
6.3	C – Multithreading Data.....	25
6.4	D – Json For Email Client.....	26
6.5	E – Git Log Master Branch.....	27
6.6	F – Meistertask Project Planning Log .....	27
7	Bibliography .....	28
8	References .....	30

---

# 1 Introduction & Overview

## 1.1 Project Overview

The Nim programming language recently entered version 1.0 as of September 23<sup>rd</sup> 2019. In the announcement blog post, the creators speak of Nim’s focus on “efficiency, readability and flexibility”.<sup>i</sup> This project was devised to test those claims directly, creating a small variety of programs focused on different areas of this claim, and to reproduce them in alike languages, measuring data, timings, and memory usage where applicable, and providing an in-depth discussion of how Nim matches up to it’s more matured language competitors.

The three areas that have been selected to compare Nim are: An implementation of the Boids flocking algorithm, focusing heavily on memory usage and profiling garbage collection, a simple implementation of a multithreaded program to calculate prime factors, with focus on language abstractions and any overhead that comes with them, and finally a small front-end web application mock-up of an email client, to showcase Nim’s ability to compile to Javascript. The choices and reasonings for the chosen projects will be discussed In chapter 2.

## 1.2 The Nim Programming Language

The Nim team describe Nim as a statically typed compiled systems programming language, combining successful concepts from mature languages like Python, Ada and Modul<sup>ii</sup>. Many of Nim’s features highlighted on its website stem from combining the pleasantries of higher-level languages, such as garbage collection, with the efficiencies and control of lower level languages, such as C compilation and macro support.

It can already be seen, then, that Nim is trying to compete over a large amount of use cases, combining the best parts of its inspirator’s features while packaging them in a singular language that can be useful for any multitude of general-use programming concerns. Especially with being a fresh language very recently debuting it’s 1.0 release, it would seem very unlikely that Nim would be able to have combine all of these disparate ideas while maintaining its target slogan of “Efficient, expressive, elegant” and the rest of this report sets out to quantize how close Nim is to that goal.

## 1.3 Aims & Objectives

The aims of this report are to compare the various qualities Nim claims to have against competitors for each of these areas. This report should be deemed successful if comparisons can be made against at least some of Nim’s core qualities with both subjective and objective evidence where required, such as garbage collection logs, timings of code blocks, and comparisons of language abstractions to complete the same goals.

---

## 2 Research

It was decided early on in planning that 3 small programs would be an adequate amount to provide a technical analysis in line with the objectives set out in chapter 1.3, with 3 opportunities to look at Nim from completely different angles.

### 2.1 Garbage Collection

Traditional memory management (such as in C like languages) is split between both the stack and heap. While stack memory can be simply described as the area of memory used for your program, that can store variables with fixed sizes known at compile-time, heap memory is slightly more complex.

Heap memory can be explained as being dynamically allocated at run time, with slightly slower access speed due to the necessity of using pointers. The benefits of this are that it can be used for variable storage with unknown and changing size requirements, for example a linked list that needs to grow and shrink due the program's lifetime. A comparison of their use cases are that the stack is used when you know exactly how much data you need before compile time and it is reasonably sized, whereas the heap is used for large allocations or allocations of unknown sizes.<sup>iii</sup>

Memory leaks are common problems in programs using traditional memory management approaches. Memory leaks are caused by memory being allocated on the heap but never freed (e.g. a `free()` call is never made in a language such as C), this can cause RAM usage of the application to increase beyond its actual requirements as the pointer to this heap memory will become lost, with no way to free.

In situations where applications are long-running and have periodic memory loss, this can culminate in a situation known as "Software aging" in which software begins to get less responsive and more error prone relative to its time running. An example of how dire these memory leaks can be shown by the findings of an experiment of software aging in the Eucalyptus cloud computing infrastructure, which found that 32 bit VM instances ran with the software would crash due to a spiralling memory leakage problem.<sup>iv</sup>

Garbage collection (GC), as described by Paul R. Wilson, is "the automatic reclamation of computer. While in many systems programmers must explicitly reclaim heap memory at some point in the program, by using a '~free' or 'dispose' statement, garbage collected systems free the programmer from this burden."<sup>v</sup> The obvious benefits of this approach is the complete lack of necessity to track both heap allocation and release, therefore almost eliminating the problem of memory leakage. The main downside, on the other hand, is the need for a constant time-consuming process to collect memory garbage. While this can be controlled in most languages with manual calls to the GC and maximum operating times during a collection, it still necessitates a large enough pause to stop it being a valid alternative for many high-performance systems.

One of Nim's more unique traits, is its ability to run with or without a garbage collector while also giving a level of customizability of the GC not normally seen in other languages. In fact, one of Nim's more elaborative document pages provides a full reference of the 7 modes the GC can run in, with extra detail for configuration, profiling, and best use-cases for each one.<sup>vi</sup>

Because of Nim's apparent push on its GC ability, memory management was chosen as a core feature to be examined in this report. Because of this choice, the attached program to test this would need heavy reliance on allocating, deallocating, and moving memory, both in large chunks at once and in small chunks but consecutively, which will be explained further in chapter 3.

### 2.1.1 Nim's GC Modes

As mentioned above, Nim has a variety of modes and settings that the garbage collector can run under, with the ability to disable it completely and to rely on manual memory allocation and de-allocation.

For example, the GC can be given specific microsecond time slots to do work, be manually called to do that work, be given differing sizes of “workPackages”, the amount of objects to be checked per stop, and a variety of profiling tools to check performance.<sup>vii</sup> Coupling this with the variety of GC strategies, such as allowing choice over whether each thread the program creates has a shared heap or spread specific heap (Mark & Sweep and Boehm strategies respectively)<sup>vi</sup>, Nim gives a greater amount of choice on it's memory strategy than most other languages, and one that can be measured and tweaked during this paper.

### 2.1.2 Valgrind's Massif

Massif is a subsidiary tool by Valgrind, creators of memory management profilers and checkers, the most famous of their tools being Memcheck, which “detects memory-management problems, and is aimed primarily at C and C++ programs”.<sup>viii</sup> Massif is a heap profiler, which works by taking snapshots of the program's heap at regular intervals. This can be directly useful to the first application of this report as “It produces a graph showing heap usage over time, including information about which parts of the program are responsible for the most memory allocations”, the graph of which would help differentiate the efficiency of GC between C# and Nim.

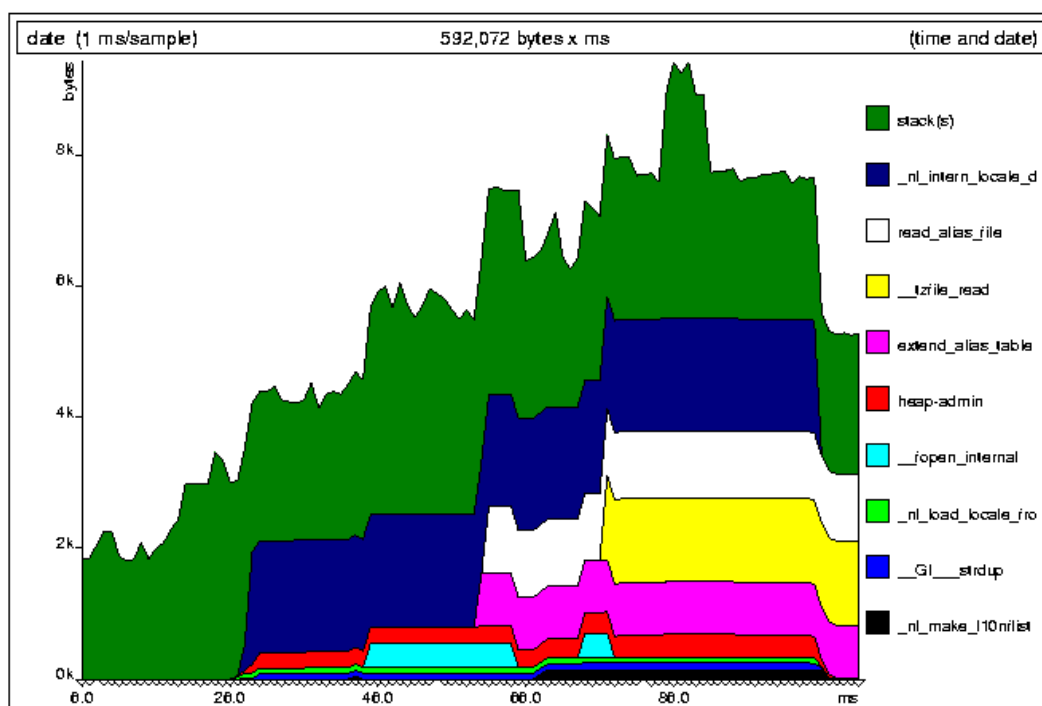


FIGURE 2.1 – MASSIF GRAPH

As seen in Figure 2.1, the Massif graph show's the amount of bytes allocated on the heap differentiated by lines of code over time. This could be helpful for analysing how different language abstractions of the same computation interact with the heap.

## 2.2 Multithreading

Multithreading is the second area of Nim in which this report will investigate and there are various obfuscations of MT that can be used, with some being standard that almost all languages will support and some, namely TPL or the Task Parallel Library being language specific (this case with C#).

### 2.2.1 Pthread & Managed Threads

Pthread's or POSIX threads, are an old standard of implementing parallelism, becoming part of the C POSIX standard to standardize MT across different hardware vendors. Pthread's are focused on being lightweight and efficient at data-exchange, largely due to the fact that they share the same heap, so memory copies are not required. Over a set of past and present hardware, it can be seen that Pthread's can be anywhere from 5 to 20 times faster than the related `fork()` process in Unix to create new processes.<sup>ix</sup> Nim contains an implementation of the POSIX standard so has access to Pthread's while C# does not.

Both Nim and C# also have access to their own threading libraries, with key differences from the Pthread's described above.

Nim's standard threads are written around GC performance, as "Each thread has its own (garbage collected) heap and sharing of memory is restricted. This helps to prevent race conditions and improves efficiency."<sup>x</sup> While this would increase productivity on creating MT applications due to the lower risks of race conditions between threads and also increase productivity of the GC due to per-thread collection, it does necessitate the need for sharing any and all data between thread's as they will share no heap that is not explicitly copied over for every change. The existence of Nim's GC modes however, allow changing the GC to a Boehm style, removing the restriction of per-thread heaps and allowing overhead free data exchange due to all thread's being able to share pointer's to the same set of memory.

### 2.2.2 C# Threading Abstractions

While Nim contains just a few different methods of handling thread's, C# as the more mature language has various abstractions from the creation and managing of threads, to data communication and locking. A few abstractions and methods that may be relevant to performance will be discussed:

### 2.2.3 Task Based Multithreading

Task based multithreading works by abstracting units of works into Tasks, that on creation return a promise, a "promise of a value (of type T) that will be available in the future, once the operations is done".<sup>xi</sup> For example a calculation that can be split at some point into 4 independent calculations could be split into 4 tasks in the form `Task<ReturnType> answer1 = new Task<ReturnType>(SomeFunction(Params));`

These 4 tasks could be created as such and then when their answer became relevant, the returned answer grabbed or waited on. This method of threading allows simple and easy abstraction, removing the overhead of the management of threads (interrupting, joining, etc.) and allowing them to be thought of as independent units of calculation with simple to access return values.

### 2.2.4 PLINQ

```
var source = Enumerable.Range(1, 10000);

// Opt in to PLINQ with AsParallel.
var evenNums = from num in source.AsParallel()
               where num % 2 == 0
               select num;
```

FIGURE 2.2 PLINQ EXAMPLE



LINQ or Language Integrated Query is a “strongly typed, logically structured syntax for querying data” that can be used to “query SQL databases, XML files, or generic data structures such as lists and queues.”<sup>xii</sup> When a subset of LINQ known as Parallel LINQ is combined with the TPL (Task Parallel Library) mentioned above, they create the Parallel Extensions Library, supported natively in .NET, which allows you to optimize your queries by splitting them into parallelly computed subqueries.<sup>xiii</sup> As seen in figure 2.2, this abstraction allows similar code written with something such as PThread’s to be massively reduced and made more understandable.

## 2.3 Data Gathering

As this project requires a large amount of both data gathering and comparison with respect to other areas of the language, looking at past projects that have attempted similar things was a requirement.

One such attempt, “completely-unscientific-benchmarks” pit an implementation of the Treap randomized binary search tree against several languages with various setups for memory management.<sup>xiv</sup> Treap is a combination of both a binary tree and a binary heap, storing X,Y pairs in a binary tree in a way that it can be binary searched by X and is a binary heap by Y.<sup>xv</sup> The Treap is known for being memory intensive, a reason given for it’s inclusion in the benchmark project, and seemingly also a good fit for the memory management section of this project. Treap was, however, not chosen as part of this project due its extremely simplistic complexity. While it would create a lot of strain on the GC of both Nim and C#, this is all it would do, and would not allow an accurate analysis of how language feature’s and abstractions could decrease/simplify lines of code or any other metric due to its simple implementation.

The benchmarking does provide another important point to take into consideration when designing the tests of this project.

5

Language	e12s	M.C.	Real Time, seconds	Slowdown Time	Memory, MB	Binary Size, MB	Compiler Version
<i>Best tuned solution</i>			0.167	x1	0.25		
Ada "naive unsafe raw pointers"	♥ (6)	♥ (8)	0.24	x1.44	0.4	0.292	GCC Ada 9.3.0
C++ "java-like" (clang)	♥ (7)	♥ (5)	0.33	x2	0.5	0.018 + libstdc++	Clang 9.0.1
C++ "java-like" (gcc)	♥ (7)	♥ (5)	0.35	x2.1	0.5	0.039 + libstdc++	GCC 9.3.0
C++ "naive unsafe raw pointers" (clang)	♥ (6)	♥ (8)	0.20	x1.2	0.4	0.014 + libstdc++	Clang 9.0.1
C++ "naive unsafe raw pointers" (gcc)	♥ (6)	♥ (8)	0.19	x1.14	0.4	0.026 + libstdc++	GCC 9.3.0
C++ "naive shared_ptr" (clang)	♥ (6)	♥ (6)	0.36	x2.1	0.5	0.018 + libstdc++	Clang 9.0.1
C++ "naive shared_ptr" (gcc)	♥ (6)	♥ (6)	0.35	x2.1	0.5	0.051 + libstdc++	GCC 9.3.0
C#	♥ (9)	♥ (1)	0.73*	x4.4	10	N/A	.NET Core 3.1

Figure 2.3

As seen in figure 2.3, various attributes were recorded from each run of the program, with different memory management strategies, with comparisons taking everything into account from memory

usage, binary size, and ‘expressiveness’ (seen here as e12s), meaning a subjective view on the productivity and ease of writing the algorithm. This is something that seems to be overlooked generally in language comparisons and it is important to know what negatives a more computationally fast language and implementation comes with.

## 2.4 JavaScript Compilation

Nim can compile its code to both front-end and back-end JavaScript. This is not a common compiler target for most languages so a portion of this project will be on investigating how complete and able this translation is, and how it compares to writing base JavaScript.

### 2.4.1 Front End

The DOM (Document Object Model) is the API for both HTML and XML documents that allows traversal and changes to the document in a tree like structure and is the main body of work that front-end JavaScript handles.<sup>xvi</sup> Nim contains a system library for the DOM that facilitates it’s compilation to JS, converting most DOM objects and functions to strongly typed versions fitting in with the Nim ecosystem, with a few additional utility functions added in.

### 2.4.2 Node JS

V8 is a C++ engine created by google that can handle the assembling of JavaScript code to run in applications it is embedded within. Its most common and well-known use case is that of being the JS engine that Google Chrome uses.<sup>xvii</sup> V8 is also used in a project called Node.JS, a JavaScript runtime built with the V8 engine that Nim can also compile for.

Node’s marketed uses are mostly for that of a back-end web server, the computational part of a web server that handle’s HTTP GET/POST parameters, required for serving of dynamic web pages, as well as handling any database communications. This is commonly referred to as an MVC framework, or Model-View-Controller.

The model part of this acronym refers to the structuring of data, for instance how a post or user of a web forum would be handled in code. The view part is the actual appearance of the data, the HTML and CSS. This may also be generated dynamically using something called a view engine, a tool that allows an amount of scripting that is interpreted into pure HTML per request, allowing dynamic content.<sup>xviii</sup>

Finally, the controller is the core of the business logic, for example facilitating everything that is required from when a user presses the reply button on a forum post to the forum post existing in the database.<sup>xix</sup>

While this back-end web server approach is the common usage for Node.js, others are also commonly spouted as a strength, such as creating API’s, microservices, and generic scripting and automation tools.<sup>xx</sup>

---

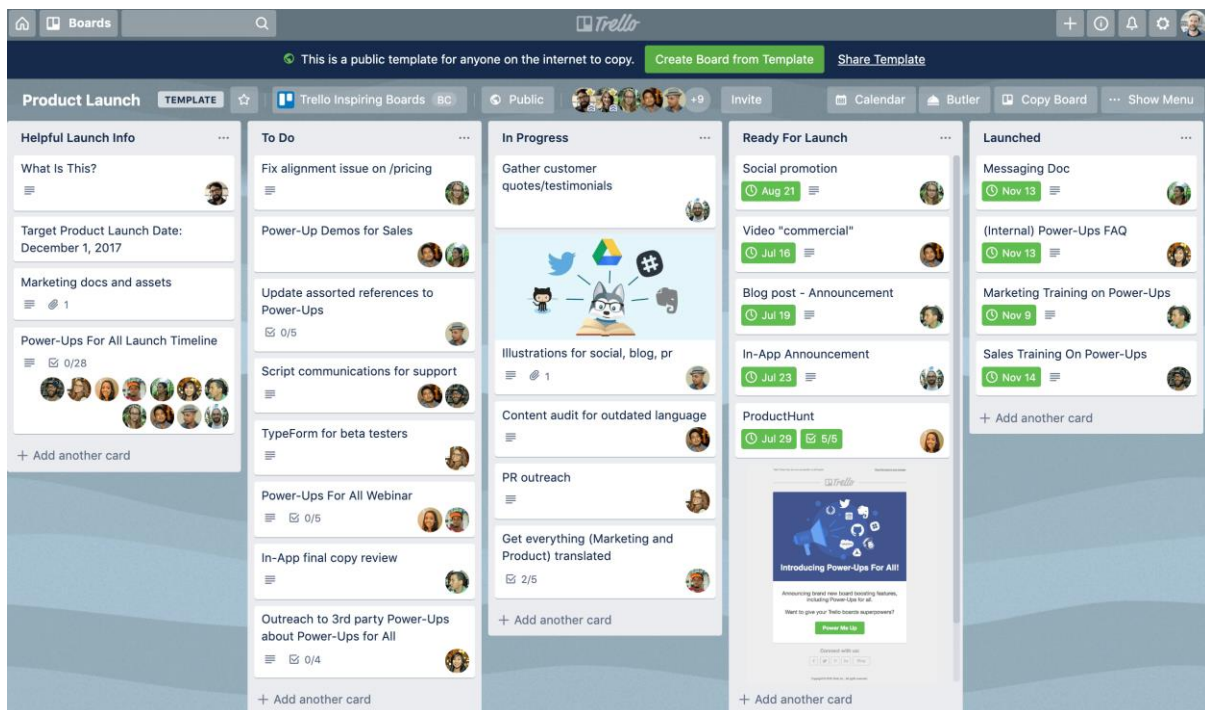
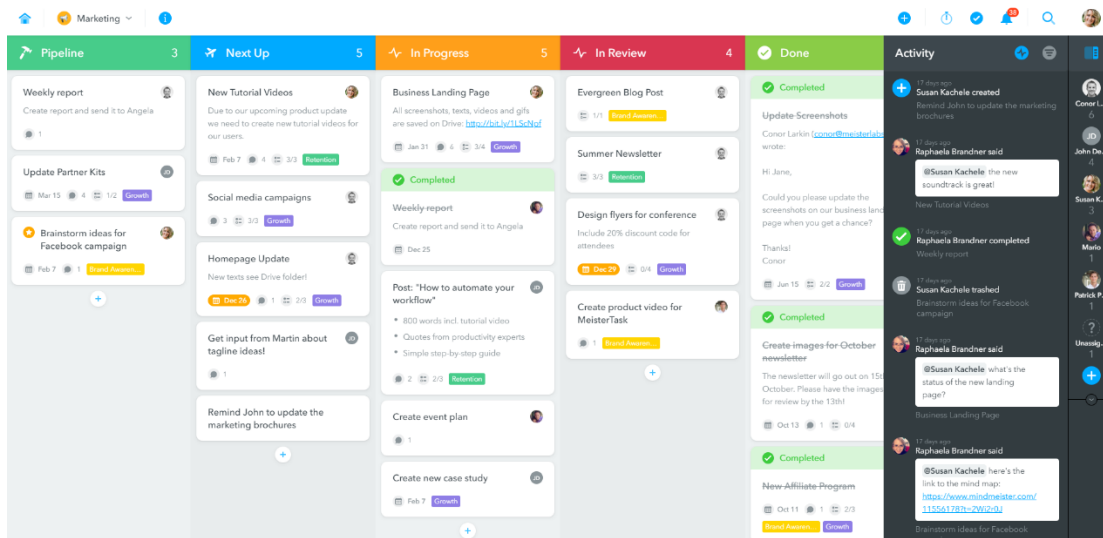
## 3 Design

### 3.1 Planning & Management

3 pieces of software were looked at for project management: Trello, Meistertask and Microsoft Team Services (MST).

As this project is focused around creating a number of small projects, the hard-reliance on agile-like planning such as differentiating tasks between Epics, User Stories, and Features, was seen as having too much overhead for small projects and MST was scrapped.

Both Meistertask and Trello are Kanban based tools, having tasks on various vertical boards organized by headers such as “To Do, Done, Testing, Bugs” etc.



Figures 3.1 and 3.2, MeisterTask & Trello respectively

MeisterTask was finally chosen for project management due to its simplicity and integration with various services such as Google Docs if I ever needed to keep a reference to a document of notes etc.

Git was also used in conjunction with GitKraken (client app) for version control, though any complex features such as branching were not necessarily needed due to the small scope of the program's and that changes could normally be made in a single file and commit without any issues. GitKraken was chosen as the Git client due to its ease of use and visualization tools better than that of the command line Git app, and the writer's familiarity with the tool.

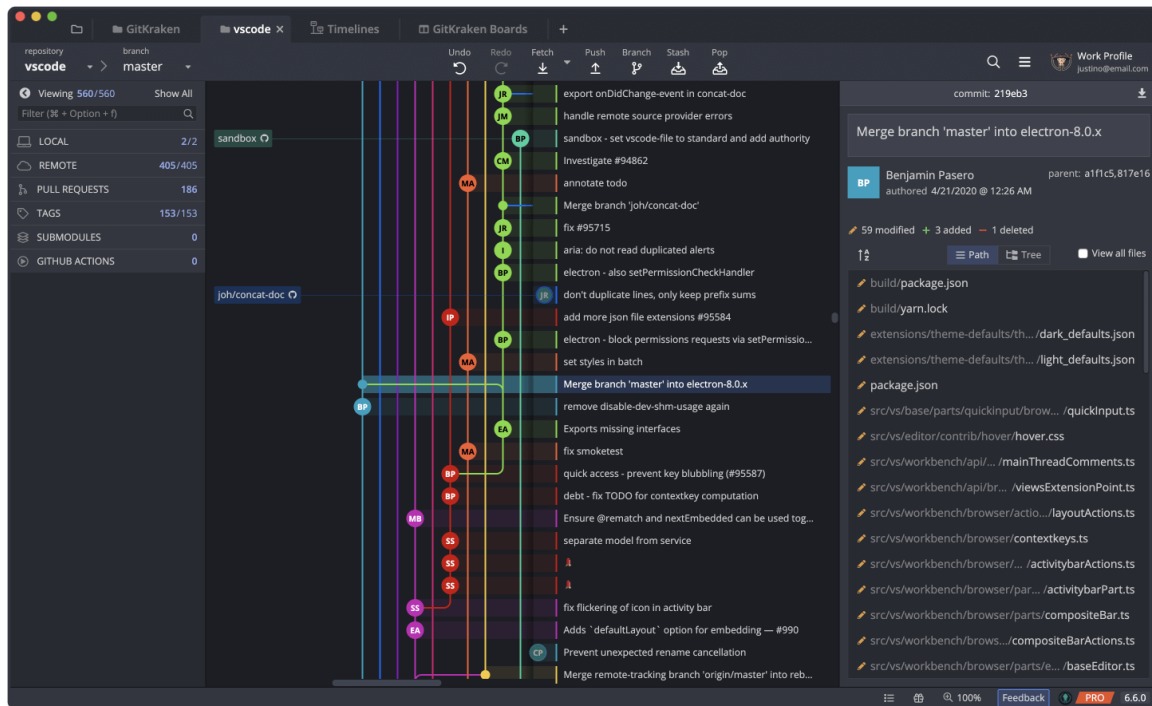


Figure 3.3 GitKraken Visualization

## 3.2 Applications

### 3.2.1 Boids

Boids is a flocking algorithm meant to simulate the movement's and co-ordinations of bird like creatures. The original paper describes the algorithm as "The simulated flock is an elaboration of a particle system, with the simulated birds being the particles".<sup>xxi</sup> Boids works by combining the aggregate position of surrounding Boids, the aggregate direction of, and a small force pushing away from nearby Boids into a single vector that is calculated per Boid and executed in a single step. While Boids on the surface does not have much to do with memory management or the GC, it is a complex algorithm that can push the GC with Boids being allocated and removed from the system at runtime in large numbers.

This combination of complex algorithm and memory stress allows a solid example to measure timings from, while also being able to compare the language abstractions used in a more realistic sense, which would not have been possible with a small dummy algorithm that just had heavy memory usage.

### 3.2.2 Prime Factors

For Multithreading, a small algorithm was picked up as a core example from Rosetta Code, a website of useful algorithms ported to a variety of languages for study.<sup>xxii</sup> The algorithm is to calculate prime factors for large numbers. The algorithm calculates prime factors for large numbers in parallel. An implementation was given for C# using one specific type of abstraction, though one would need to be made for Nim, and other C# varieties produced for better comparisons above just CPU times.

### 3.2.3 Web Mail Client

Due to the discussion in chapter 2.4 about Nim's compilation to JavaScript, the decision was made to create a small JavaScript heavy dummy web app, in this case posing as a web mail client, akin to Gmail. Back-end implementation ideas were scrapped due to the amount of boilerplate code required by Node.js such as setting up back-end MVC web server frameworks as discussed. A front-end application could be made with minimal useless code (a small amount of HTML/CSS) and

JavaScript would be under heavy usage in a web client due to the amount of UI required. The web client will also read a list of emails from a JSON file as dummy test data to confirm everything is working correctly.

---

## 4 Implementation

### 4.1 Boids

#### 4.1.1 Code

The first thing needed for Boids would be a graphics library to display the Boids for visual testing. SFML was chosen due to its popularity as a relatively simple 2D Multimedia library. An additional benefit of using SFML is that it has bindings for both Nim and C#, so can be used in both projects with minimal changes to the code structure and thus minimal effect on performance monitoring.

Code Layout in Nim was done via 2 files, an *index.nim* for logic and a *funcs.nim* for function storage. As C# does not have the C-like requirement of having functions declared before calling, the C# version simply combines both into one file.

Initial setup for SFML consists of creating a video mode, list of render settings and then creating a window with this information. From that point sprites and shapes can be moved and drawn with a relatively simple API. See figure 4.1 for SFML setup.

```
#region constants
static readonly Texture bird_texture = new Texture("triangle.png");
static Vector2u sz = bird_texture.Size;
static VideoMode videoMode = new VideoMode(1000, 1000);
static ContextSettings settings = new ContextSettings { DepthBits = 32, AntialiasingLevel = 32 };
static RenderWindow window = new RenderWindow(videoMode, "Boids - Liam Warner", Styles.Default, settings);
```

Figure 4.1 SFML setup in C#

The window is opened as soon as the RenderWindow is created, initially opening a blank window with the background cleared to black. From this point the core loop of the program can begin, held within a *while window.open*.

As this program is essentially a wrapper around data collection, a way needed to be created to differentiate between testing Boids as a behaviour, and profiling Boids as a unit of computation. This was achieved using Nim's default ParseOpt commands and C#'s more expressive OptionSet class.

```
var doProfiling = true
var p = initOptParser("")
while true:
  p.next()
  case p.kind
  of cmdEnd: break
  of cmdShortOption, cmdLongOption:
    if p.key == "justBoids" :
      doProfiling = false
    else : break
```

```
bool doProfiling = true;
var optionSet = new OptionSet()
{
  { "j|justBoids", v => doProfiling = false }
};
```

Figure 4.2, Command option parsing in Nim and C# respectively

As seen in figure 4.2, reading simple command line instructions in Nim requires a while loop, a switch statement, and manually checking of both short options (*-a*) and long options (*--doSomething*). Meanwhile C# required just an initializer list for a class while their implicit constructor

handles the manual work needing to be written in Nim. This is the first of a reoccurring theme of C# providing essentially the same logic with vastly reduced code and improved maintainability.

After this initial setup of SFML and reading the correct mode (data collection or just running the Boids algorithm), some setup for the Boids algorithm is done. In both C# and Nim this consists of some fairly standard creating of a dynamic list to hold the quantity of Boids to be created, as well as creating said Boids. Each Boid is generated randomly with the `MakeBoid()` function shown in figure 4.3, being placed and rotated randomly within the screen space.

```
proc MakeBoid() : Sprite =  
  var bird = new_Sprite(bird_texture)  
  bird.origin = vec2(sz.x/2, sz.y/2)  
  bird.scale = vec2(0.01, 0.02)  
  bird.position = vec2(rand(window.size.x), rand(window.size.y))  
  bird.rotation = rand(360f)  
  bird.color = color(150,150,0)  
  return bird
```

*Figure 4.3 creating a random Boid*

Various helper functions exist around the `MakeBoid` command and the array of active Boids such as Adding a set number using a for loop, and removing them from random positions in the dynamic list (Sequence in Nim) while moving indexes accordingly to leave no gaps, implicit behaviour in both C# and Nim.

Under normal circumstances (not data collection mode), the core loop, besides handling window exit commands, contains just 2 functions. The first, `DoBoids`, holds all the logic for the actual Boids algorithm which when done will be drawn to the screen.

In `DoBoids`, each Boid is looped through and their projected movement calculated, this movement is then transformed into a rotation using the formula  $\text{Degrees} = \arctans2(\text{move.x} - \text{move.y}) * 180/\pi$  to rotate the Boid facing the direction it is moving in. Then each Boid is moved this direction with respect to Delta Time (the time difference between the last and current frame, to maintain speed regardless of framerate).

The actual calculation of movement, as described by the referenced Boids paper, is a combination of:

- The aggregate position of all nearby Boids (in a defined “vision” radius).
- The aggregate alignment of all nearby Boids (in vision radius).
- A negative vector away from any Boids inside an avoidance distance (a fraction of the vision radius in this case)

Each of these calculations are then multiplied by a weight (to allow tweaking of each movement’s influence) and normalized if the square magnitude of their vector is greater than the squared weight. Both the calculation and normalization can be seen in figure 4.4

```

proc SqrMagnitude(vec : Vector2f) : float =
    return vec.x * vec.x + vec.y * vec.y

proc NormalizeTimesWeightIfOver(vec : Vector2f, weight : float) : Vector2f =
    var returnVec = vec
    if(SqrMagnitude(vec) > weight * weight):
        returnVec = vec / SqrMagnitude(vec)
        returnVec = returnVec * weight
    return returnVec

proc CalculateBoidMove(currentBoid: Sprite, boids : seq[Sprite], i : int) : Vector2f =
    let nearby = GetNearbyBoids(currentBoid,boids,i)

    #Cohesion Move
    var cohesionMove = vec2(0f,0f)
    for i, cohesionBoid in nearby:
        cohesionMove = cohesionMove + cohesionBoid.position
    cohesionMove = cohesionMove/(float)nearby.len
    cohesionMove = cohesionMove - currentBoid.position
    cohesionMove = cohesionMove * WEIGHT_COHESION
    cohesionMove = NormalizeTimesWeightIfOver(cohesionMove,WEIGHT_COHESION)

    #Alignment Move
    var radians = 2 * PI / 360 * currentBoid.rotation;
    var alignmentMove = vec2(0f,0f)
    if(nearby.len == 0):
        alignmentMove = vec2(sin(radians),-cos(radians))
    else:
        for i, alignmentBoid in nearby:
            radians = 2 * PI/360 * alignmentBoid.rotation
            alignmentMove = alignmentMove + vec2(sin(radians),-cos(radians))
        alignmentMove = alignmentMove / (float)nearby.len
    alignmentMove = alignmentMove * WEIGHT_ALIGNMENT
    alignmentMove = NormalizeTimesWeightIfOver(alignmentMove,WEIGHT_ALIGNMENT)

    #Avoidance Move
    var avoidanceMove = vec2(0f,0f)
    var boidsToAvoid = 0
    for i, avoidanceBoid in nearby:
        if(SqrMagnitude(avoidanceBoid.position - currentBoid.position) < BOID_DISTANCE * BOID_DISTANCE * BOID_AV)
            boidsToAvoid = boidsToAvoid + 1
            avoidanceMove = avoidanceMove + (currentBoid.position - avoidanceBoid.position)
    if(boidsToAvoid > 0):
        avoidanceMove = avoidanceMove / (float)boidsToAvoid
    avoidanceMove = avoidanceMove * BOID_AVOIDANCE_MULTIPLIER
    avoidanceMove = avoidanceMove * WEIGHT_AVOIDANCE
    avoidanceMove = NormalizeTimesWeightIfOver(avoidanceMove, WEIGHT_AVOIDANCE)

    #Final Changes
    var finalMove = (alignmentMove + cohesionMove + avoidanceMove ) / 3.0f

    if(SqrMagnitude(finalMove) > BOID_MAX_SPEED * BOID_MAX_SPEED) :
        finalMove = finalMove/SqrMagnitude(finalMove)
        finalMove = finalMove * BOID_MAX_SPEED * BOID_MAX_SPEED

    return finalMove

```

Figure 4.4 Boid calculation and vector normalization

Various helper functions were created to help with regularly occurring tasks, such as GetNearbyBoids and SqrMagnitude, the first of which just looping through each Boid in the sequence and comparing the Euclidian distance to that of the Boid's vision range. The end result can be seen in Appendix A.



### 4.1.2 Data Measuring

The data collection mode of the programs adds some extra code, namely adding and removing chunks of Boids per loop of the program, looping a set amount of times before finally flushing all the recorded times to a text file and exiting.

The program runs several main cycles of which the main timings are taken, these main cycles are repeated 75 times to allow a good number of samples to come to averages across all records. Inside each cycle is then a sub cycle that runs 300 times, each sub cycle adding 10 Boids to the system and removing 5 (at random positions). These numbers were chosen after an initial test with the knowledge in mind that these 75 main cycles would again be repeated for different GC options.

The core method for these timings is identical across the 10 different things being profiled, as seen in figure 4.5. The relevant clock is first restarted, the function is ran to completion, and then a stream to a text file is written on a new line with the microsecond period.

```
#Start of Cycle
discard addClock.restart
AddBoids(boids,boidMoves)
addBoidsStream.writeLine(addClock.restart.asMicroseconds)
```

*Figure 4.5 Clock timings of adding Boids to the system*

SFML includes a Clock class that works well for timing microsecond periods, and with the ability to use it for both the C# and Nim variant, it was used almost entirely for all time recording.

While timings were the majority of data collected, including: adding Boids, removing Boids, setting up arrays on each cycle, sub cycle timing, main cycle timing, clearing arrays at cycle end, and time taken to perform movement calculations, other profiling was also done, such as checking occupied heap memory at various points in the program's execution, mostly for GC comparison.

### 4.1.3 Findings

All findings are displayed in full, from an Excel sheet in Appendix B. For this section, the appendix shown in 6.2.1 will be used as B.1 etc. When not stated in the side bar, results are given from top to bottom as Maximum value, minimum, and average and are colour coded from best (green) to worst (red).

As timings and records were taken per sub cycle and per cycle, it gave a minimum of 75 re-runs per timing, with that increasing to 75\*300 for timings per sub cycle. These timings were also taken in as optimal a situation could be made, with minimal background CPU activity.

The overall findings in B.1 show that the best overall GC mode is the default mode, working well in all areas and an average speed in adding Boids. C# is also much slower as would be expected from intermediate code running on C#'s Common Language Runtime as opposed to the native machine code finally compiled by Nim (after it's translation to C, in this case). C# finishing first place in speed for adding is most likely due to the initial creation of the C# List allocating enough starting memory for the needs of the entire list added together with any compiler optimizations that Nim may not be taking use of.

Even looking at B.7, the difference of completion time between the various GC modes is so minimal most types, only with C# lagging behind a large 563% amount compared to the quickest method Mark and Sweep.

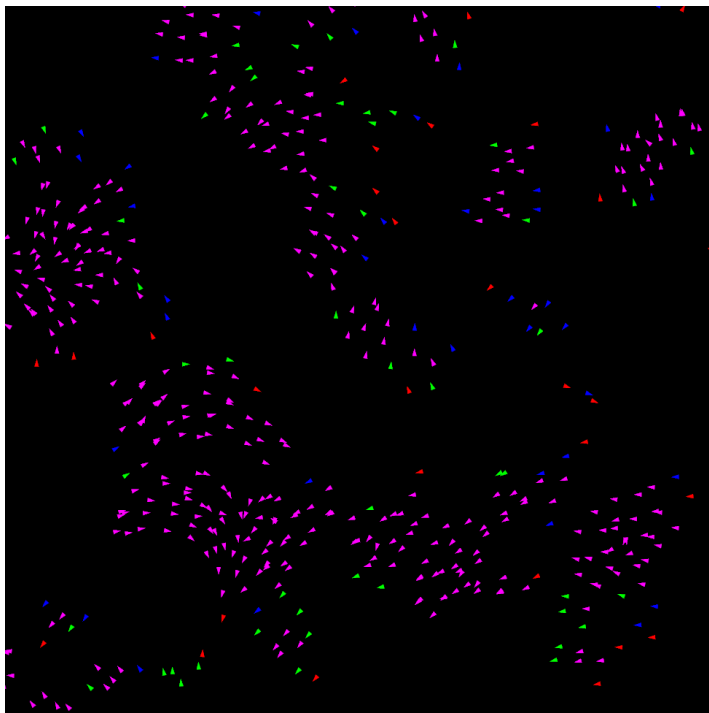
There are some paradoxical results among the data concerning C#. The biggest of these being C#'s quickest time for calculating Boid movements, being 20 us compared to the average sitting around 762. While this was not investigated fully, an educated guess would put this down to an error where the first calculation in the C# version for this particular run took place before the rest of the Boids were added to the sequence.

Impressively, when comparing program execution time for Nim with and without GC, it was found that running with GC only adds roughly a 10% overhead when adding new Boids as seen with B.2 and paradoxically running with GC active gave the system an overall boost in performance, however this is extremely minor when compared in their averages.

A downside of using a cross platform library such as SFML amongst its many positives for this particular experiment is the uniformity of code it creates between both C# and Nim as everything from the graphics code to the timing is taken from the library. For this reason, very little could be ascertained in relation to how each language's features and abstractions could have improved performance or expressiveness, and the second application will be more focused on this aspect.

#### 4.1.4 Testing

As the Boids algorithm was mostly dummy calculation to record, heavy testing was not needed further than visual checks that each application was performing the same work. To aid this a helper function named `ColourBoids` was created, that colours Boids in a switch case based on the amount of neighbours they have.



*Figure 4.6 result of the `ColorBoids` function*

While `ColourBoids` was useful in testing, it was also useful in development for fine-tuning the vision range of the Boids with graphical feedback. Currently it can only be enabled with hard-coding and no command line option exists.

## 4.2 Multithreading

### 4.2.1 Code

Multithreading was another factor chosen to be tested and a simple program was made that calculates prime factors of a set of numbers, each in its own thread. The core computation for the factors is located in `calculatePrimesOneThread` in both Nim and C# and some subtle changes exist between the two languages as seen in figure 4.6

```

static List<long> CalculatePrimes(long number)
{
    Stopwatch clock = new Stopwatch();
    clock.Start();
    List<long> primeNumbers = new List<long>();
    for (int i = 2; i < number; i++)
    {
        while (number % i == 0)
        {
            primeNumbers.Add(i);
            number /= i;
        }
    }
    clock.Stop();
    individualCycles.WriteLine(clock.ElapsedTicks / (Stopwatch.Frequency /
    Console.WriteLine("Completed");
    return primeNumbers;
}

```

```

proc calculatePrimesOneThread(num: int64, threadID: int, timeHolder: ptr BiggestFloat): seq[int64]
time(elapsed):
  var number = num;
  for i in 2 ..< number:
    while (number mod i == 0):
      result.add(number)
      number = number div i

  stdout.writeLine "Thread ", threadID, ", elapsed: ", elapsed
  stdout.writeLine "Completed"
  timeHolder[] = elapsed

```

Figure 4.6 Prime calculation in C# and Nim respectively

The main difference between the two is how the timing is done; While C# uses the Stopwatch (from System.Timers), I have taken advantage of Nim's extensive macro capabilities to create the time macro as seen in 4.7.

```

template time(elapsed, body: untyped) =
  let start = getMonoTime()
  body
  var elapsed = BiggestFloat(inNanoseconds(getMonoTime() - start)) / 1_000_000

```

Figure 4.7 Nim's macro capabilities used to create a timer block

The macro simply wraps a block that is indented the same as a conditional block (as seen in 4.6) with a timer beginning before the block of code, and converting the time from nanoseconds to microseconds after the block, then storing it in a variable. This is mostly possible due to the ease of writing Nim macros as they use basically the same structure and syntax as standard Nim code, making it easy to write and readable.

The actual multithreading for the calculations is done on a list/sequence of 64-bit integers, and this is where one of the biggest differences between the Nim and C# code is shown. Nim uses a common approach of spinning up threads, with the added nicety of being able to return a sequence of values (1 long for each prime factor) directly from the thread. The only thing that does need to be accessed via pointer is the array of values for individual thread timings. C# on the other hand, while not taking use of any macros for timing, does manage to condense the 8 or so lines in Nim to a single line of PLINQ (the SQL meets parallelization syntax discussed in the research section) as seen in 4.8.

```
List<long> toCalculate = new List<long> {
    13548153810,
    24265915380,
    35143815420,
    41452967940,
    52725472560,
    63547825480,
    74936756240, };
Stopwatch clock = new Stopwatch();
clock.Start();
var results = toCalculate.AsParallel().Select(CalculatePrimes).ToList(); //
clock.Stop();
```

```
const toCalculate = [13548153810, 24265915380, 35143815420,
                    41452967940, 52725472560, 63547825480,
                    74936756240]
var start = getMonoTime()
# Result initialization
var futureResults = newSeq[Flowvar[seq[int64]]](toCalculate.len)
timeElapsed = newSeq[BiggestFloat](7)

time(overallTime):
  for i in 0..6:
    var number = toCalculate[i]
    futureResults[i] = spawn calculatePrimesOneThread(number,i,ad
  sync()
  for i in 0..6:
    results.add(^futureResults[i])
  for i in 0..6:
    individualCycles.writeLine(timeElapsed[i]) #Move out of timing
```

Figure 4.8, the parallelisation done in C# and Nim respectively

In most languages such as C++ this would normally require a level of locking and atomic functions, passing a pointer to an index within the list and then filling it as needed from inside the thread. Here this is simplified by using the `AsParallel()` function to multithread a LINQ query using implicit C# thread pools, then running each number through the `CalculatePrimes` function, then collecting all those results into a `List` where each result's index matches the index of the original number. This abstraction even goes further as by utilizing the C# thread pool, the actual amount of physical threads being run at once is handled by the CLR and does not need to be calculated beforehand. The specific usage of the `Spawn()` command in Nim also spins up threads by using the thread pool also.

Nim handles the abstraction of directly returning values by using the Nim type `FlowVar`. While documentation on what exactly a `FlowVar` is, is very thin, an educated guess would place it as using some implicit promise like structure (as seen in the Task based multithreading of C#). This is mostly inferred due to many of the functions from the `ThreadPool` library in Nim containing functions such as `blockUntil()` and `isReady()`<sup>xxiii</sup> implying there is an internal state to the `FlowVar` signifying whether the value has been returned by the thread at a given time.

After initial calculation, some queries are ran to grab the lowest and highest values of the factors and the numbers of factors for each 64-bit integer. C# does this masterfully via the use of LINQ mentioned previously, compounding each query into a single, easy to understand line using lambda functions to supply a comparison function. As Nim does not come with any in-built querying library,

a macro was made to best represent this. Both the C# LINQ and Nim macro and usage can be seen in figures 4.9 and 4.10

```
minNum = results.Min(x => x.Count);
maxNum = results.Max(x => x.Count);
smallest = results.Min(x => x.Min());
largest = results.Max(x => x.Max());
```

Figure 4.9 C# use of LINQ for querying the list of results

```
template minMaxImpl[T](a: openArray[T], cond, atom: untyped): untyped
let
  arr = a
  var result = block:
    let it {.inject.} = arr[0]
    atom

  for i in arr.low.succ .. arr.high:
    let it {.inject.} = arr[i]
    let eval = atom
    if cond(eval, result):
      result = eval
  result

template minIt[T](a: openArray[T], atom: untyped): untyped =
  a.minMaxImpl('<', atom)

template maxIt[T](a: openArray[T], atom: untyped): untyped =
  a.minMaxImpl('>', atom)
```

```
for i in 0..6:
  var curSmallest = results[i].minIt(results[i].len)
  var curLargest = results[i].maxIt(results[i].len)
  if(curSmallest < smallest):
    smallest = curSmallest
  if(curLargest > largest):
    largest = curLargest
```

Figure 4.10 Nim macro implementation for querying

While not as expressive or easy to understand as C#'s LINQ, it does show the expressiveness of the macro abilities of Nim that this was able to be created without much background knowledge or experience with macro creation, and does simplify the querying by a large amount than it would be with just pure Nim code. However it is still not as expressive as LINQ as there still needs to be a loop and cached variable in finding the lowest/highest number whereas C# can accomplish this by running a nested query `results.Min(x => x.Min())`.

In backup-2.Nim, an approach was tested in Nim very syntactically similar to the final PLINQ C# approach in the form of `let results = toCalculate.mapIt calculatePrimes it`,

Doing as the C# version and applying a function to each individual value in a sequence and creating an output array matching the index. However this was a single threaded approach and currently no way to extend this to parallel processing as the C# version, so was scrapped to the manual threading approach that was finally used.

### 4.2.2 Findings

Measuring for this application is mostly described above as it is more linked with the application code in this example, but on averaging out the time values by doing multiple runs as shown in the first example. The results in appendix C shows a small but quantifiable performance increase in the C# PLINQ implementation over Nim while also having the C# example being subjectively more expressive and readable.

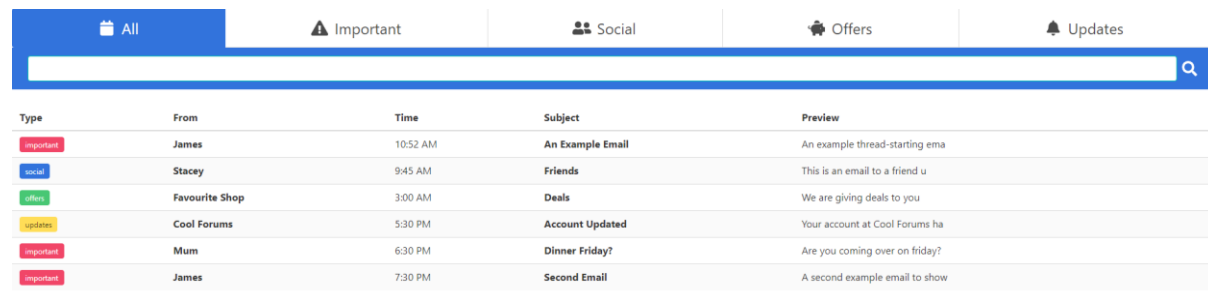
As seen in the code, there is also a commented region in the C# code showing a Task based approach to the problem, but after some initial timings it was found slower than the PLINQ approach, so the PLINQ approach was used in the comparison to Nim.

## 4.3 Web Client

### 4.3.1 Code

The mail client was produced as an application that could quickly be created that contained enough JavaScript activity to be a good comparison to the Nim compilation. The HTML and CSS uses a simple and very productive framework known as Bulma.<sup>xxiv</sup> Bulma was chosen due to being a very opinionated framework, meaning many decisions are made by the framework to remove the programmer needing to add boilerplate CSS code for many common uses. An example of this is a mobile-responsive layout system containing of grid's and human-readable classes such as "is-half" for a column to take up half the available space inside its container.

Using Bulma and a very small amount of custom CSS (mostly display properties for JavaScript to trigger via user input) a very basic HTML layout was created as seen in figure 4.11.



Type	From	Time	Subject	Preview
important	James	10:52 AM	An Example Email	An example thread-starting ema
social	Stacey	9:45 AM	Friends	This is an email to a friend u
offers	Favourite Shop	3:00 AM	Deals	We are giving deals to you
updates	Cool Forums	5:30 PM	Account Updated	Your account at Cool Forums ha
important	Mum	6:30 PM	Dinner Friday?	Are you coming over on friday?
important	James	7:30 PM	Second Email	A second example email to show

Figure 4.11 User Interface of the mail client

The layout is split into 3 main sections, the initial Navbar and search box at the top makes use of Bulma's Tabs element, allowing for multiple box like selectors to be automatically sized.

The next section of email information uses a table that is dynamically filled in by JavaScript that will be explained later.

The final section of the interface is the actual email reader shown in figure 4.12 that is manually displayed and populated using JavaScript.

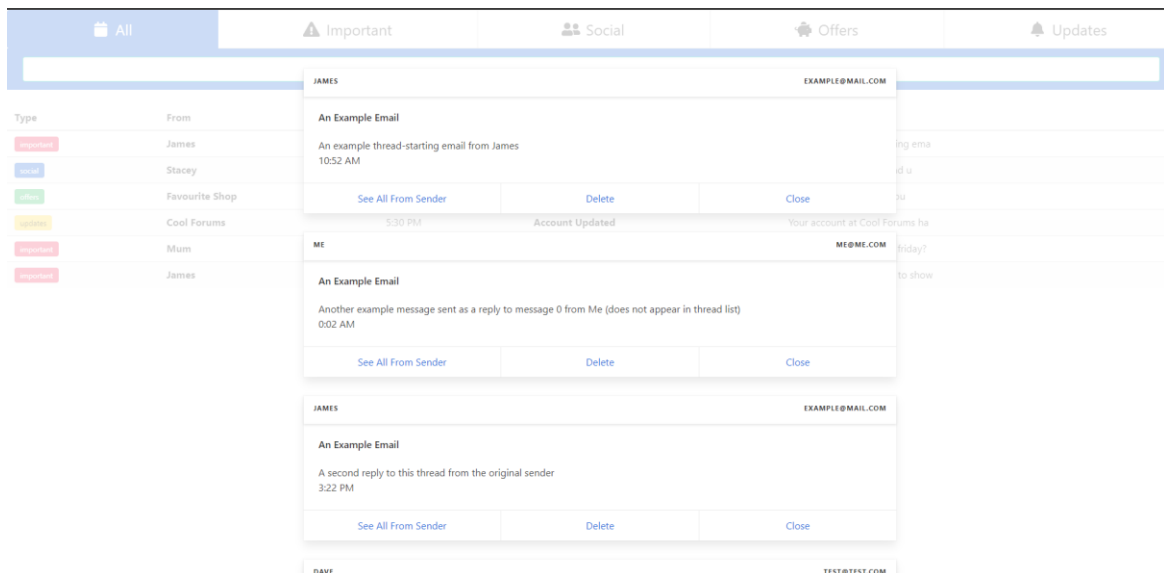


Figure 4.12 Email reader box

The email reader section shows a thread contained the initial email and any back and forth replies by both the user and recipient akin to the way Gmail shows its emails. A small opacity change is also added to the main interface to better differentiate the two.

In terms of JavaScript, the list of emails is represented by a JSON string embedded directly inside the HTML. This is done as reading from a file would require user input to insert the file as JavaScript enforces client security to stop programmatic reading of local files. The other alternative to this would be a splash screen asking for the JSON file to be uploaded, however in comparison the embedded approach is much cleaner and easier to debug due to not having to read the file first and then convert to a JSON object. A snippet of the final JSON can be seen in Appendix D. Of note is the “from\_id” which is used as an indicator of the email ID that is being replied to, this is then traversed until a null is found to create the thread like structure as seen in figure 4.12. The “type” field is also used for colour coding the tag box seen at the very left of each email in figure 4.11 and is how the filtering system works seen in the navbar.

The JavaScript code itself works in a slightly functional programming manner, with heavy use of pure functions, meaning most functions do not edit the state of the JavaScript variables, as most are assumed to be semi-constant such as the emailJson variable which holds the master copy of the email data.

This email data is then passed through to the ParseJson or ParseJsonFiltered function that handles any querying needed (e.g by tag type, sender email, or containing text input into the search bar). If filtering is needed, the mutated copy of email data is then passed to ParseJsonCustom, an identical copy of ParseJson except that the custom version takes in a parameter whereas the main ParseJson works off the master copy. These functions will then generate the HTML required to place them in the main interface. ParseJson and ParseJsonCustom would have been implemented as overloaded functions but JavaScript does not support that feature, however Nim does so would have been a good example of Nim’s feature-set expanding beyond a compared language, which has been rare so far during this project. An example of the pure ParseJson function can be seen in figure 4.13

```

function ParseJson() {
  const start = document.getElementById('mailBody')
  start.innerHTML = ""; //Clear
  Array.prototype.forEach.call(emailJson.emails, element => {
    if(element.from_id != null)
      console.log("Skipping")
    else {
      var tagType
      switch(element.type) {
        case "social":
          tagType = "link";
          break;
        case "updates":
          tagType = "warning";
          break;
        case "important":
          tagType = "danger";
          break;
        default:
          tagType = "success"
          break;
      }
      let htmlCode =
`<tr id=mail-${element.id} onclick=DisplayThread(${element.id})>
<td><span class="tag is-${tagType}">${element.type}</span></td>
<th>${element.nickname}</th>
<td>${element.time}</td>
<th>${element.subject}</th>
<td>${element.message.substring(0,30)}</td>
</tr>`
      document.getElementById("mailBody").innerHTML+=htmlCode;
    }
  });
}

```

Figure 4.13 Use of the `forEach` prototype for more expressive looping of the email JSON data.

Template strings (depicted by ```) are also a great language feature for more complex string data, allowing embedded variable output if wrapped in `${}`. Template strings are also implemented inside Nim but have more restrictions, such as being parsed as string literals, meaning a `"\n"` to depict a new line will be printed character for character.<sup>xxv</sup>

Some other small JavaScript functions are implemented such as the JSON being loaded specifically after the body. This is required due to the JSON being embedded directly into the HTML, requiring the entire document be loaded before any JavaScript should be run. CSS class based functions are also created such as `"SetActive"` for swapping the active tabs in the navbar and re-populating the email list based on the chosen tag.

The most complex piece of JavaScript present is `DisplayThread()` which takes in an email ID (as seen in appendix D) and loops through the mail list for a separate email with the corresponding `"from_id"`. This is repeated until no emails can be found as replies, then the HTML is generated along with some buttons on each email allowing further filtering by the sender of the email.

### 4.3.2 Findings

During porting of the original JS code to Nim, it was discovered that some core features were missing from the Nim DOM library. Necessary utility functions such as getting the parent of a DOM element are completely missing, however some fancier features exist such as `SetDragImage()`. At the discovery of this, the completed Nim implementation would likely be not comparable as core logic changes would need to be implemented for any resemblance of a functioning app akin to the pure JS app. For this reason, a Nim implementation has not been completed also due to the large amount of additional time required to not only code the entire thing from scratch but also for making any necessary design changes.



---

## 5 Evaluation

### 5.1 Data Recording

The project has met most of its aims and objectives such as making comparisons of Nim to other language using data where required. The data involved in some of these comparisons is not ideal, with various erroneous points and subtle differences between the versions of code further adding differences not intended to exist. Examples of erroneous data include the better performance seen with GC enabled in Appendix B.2. As different libraries and functions were used for both recording and outputting data to files, there is an unknown amount of overhead added to each system that has not been fully investigated, adding bias to each recording. For example while the text streams are manually flushed outside of the main code blocks being timed, data was outputted to text files during execution in some cases, meaning automatic flushing (giving non-trivial I/O overhead) is happening that has not been accounted. Even more than that this overhead is not identical between the two versions of each application as their stream libraries are different.

An improvement to this would be to have researched a cross-platform library to use for both measuring and file streams, preferably with manual flushing to avoid IO overhead during collection. The timers also have slightly different resolutions, and while standardized around microseconds in the Excel spreadsheets, Nim's `getMonoTime` function returns a time period in nanosecond resolution however the `stdlib Clock` used in C# is in microseconds, creating different accuracies between the two.

Random placement of Boids could also have been approved as using each language's internal PRNG creates differences between both runs and applications, and as computation time can vary wildly based on Boid placement, this has skewed results and is a reason why such a large amount of averages were taken. This could be easily resolved by using a cross-platform PRNG or just supplying the program with a pre-made set of locations and rotations to each program to create a more identical process.

Although some failures happened in data collection, overall it can be seen as successful as at least certain parts of the data collection are valid, such as comparisons between different GC modes for Nim, and the complete time's taken to run the Boids program, and although background usage was controlled in a minor way, there will still be some minor differences.

### 5.2 Organization

The decision to split the project into creating multiple small programs rather than a large multi-use monolithic one was a good one. It allowed easier tracking of git logs (seen in Appendix E) for per-project changes. It also led to better code readability and maintainability for changes as each program was its own modular system. This also helped with overall project planning (see appendix F for MeisterTask log) as having smaller projects made splitting them into sub-tasks much easier.

### 5.3 Nim

One of the main concepts to be explored throughout this project was the usability and "business-readiness" of Nim as a recently launched language. While Nim did provide some effective language features such as good macro support, it was severely lacking in common abstractions such as the Task based programming of C# and better alternatives to parallelise functions running on arrays. The weakness of an incomplete DOM library with no method to fetch parent elements is perhaps the best indicator of this. During development, even some compiler errors were misleading beyond any common error that would be found in, for example, the C# CLR compiler. An example of this was

receiving an “Array index out of bound” error on trying to access an uninitialized array. More mature languages would report on the uninitialized variable above trying to access an out of bounds index.

This unhelpful reporting is even greater with JavaScript compilation;

```
C:\Users\rogue\Desktop\Wim Test\Javascript\Wim\index.nim(4, 5) Hint: 'tabs' is declared but not used [XDeclaredButNotUsed]
C:\Users\rogue\Desktop\Wim Test\Javascript\Wim\index.nim(7, 6) Hint: 'SetActive' is declared but not used [XDeclaredButNotUsed]
C:\Users\rogue\Desktop\Wim Test\Javascript\Wim\index.nim(3, 5) Hint: 'emailjson' is declared but not used [XDeclaredButNotUsed]
C:\Users\rogue\Desktop\Wim Test\Javascript\Wim\index.nim(1, 8) warning: imported and not used: 'json' [UnusedImport]
Hint: 38432 LOC; 0.338 sec; 37.848MiB peakmem; Release build; proj: C:\Users\rogue\Desktop\Wim Test\Javascript\Wim\index.nim; out: C:\Users\rogue\Desktop\Wim Test\Javascript\Wim\index.js [SuccessX]
Hint: "C:\Users\rogue\Desktop\Wim Test\Javascript\Wim\index.js" -o: indexN.js [Exec]
oserr.nim(94) raiseOSError
Error: unhandled exception: The parameter is incorrect.
[OSError]
PS C:\Users\rogue\Desktop\Wim Test\Javascript\Wim>
```

Figure 5.1 JavaScript Compilation Error

The above figure 5.1 shows a generic error of “The parameter is incorrect.” On compilation of JavaScript code. This error would not be too out of place if a line number were also given to the correct bit of user code in which it occurred. However, it seems as the JS compiler does not climb the call stack to report the user code line, but simply flags the file and line of where the exception happens, in this case `asser.nim`, a system file. This would mean that to properly debug this very common error message, a full stack trace would be needed explicitly to find where the error occurred in user code.

## 5.4 Incomplete Projects

Various methods for improving and fixing errors in the applications have been given, and would not have taken an extraordinary amount of time to implement, but due to timing difficulties largely because of unexpected waiting time to record all the data gotten, many of them have gone unresolved as well as some minor bugs such as the Boids turning instantly when they should have a lerped rotation. In hindsight, recording less examples for averages would have potentially freed up the time to fix most if not all these smaller issues.

## 5.5 Personal Development

Personal development was prominent throughout this project, with Nim being an unfamiliar language to me at the start besides knowing a few of its main features. Syntax, libraries, and macro writing was learned throughout this project, and large amount of reading through Nim’s manual and guides was required due to find any language abstractions and features relevant to the applications. An example of this is the `FlowVar` promise based type used in the MT application for returning data from threads. While C# and JS were known by me at the beginning of this project fairly well, there was also a need to explore them more deeply for abstractions I was not aware of such as the C# `PLINQ` used in application 2.

As well as development of language syntax and features, I read about the technical workings of how to best measure and present data. This led me to use many averages of the same timing to increase reliability along with attempting to only manually flush data streams after recording was over due to the I/O overhead (although this was unsuccessful).

There was also a skill developed from scratch in porting programs, as each application was developed fully in one language before then being ported to the other, looking for opportunities to use language-specific features where possible. This area shows as a weakness throughout the project, as many of the applications are not as close representations of each other as they should be, with some differences affecting data recording that shouldn’t be there (the aforementioned issue with flushing data to .txt files during execution).

Project management and Git usage was also lower than it really should have been for a project of this scope. Although attempts were made to make each git commit modular for easy roll-back, by

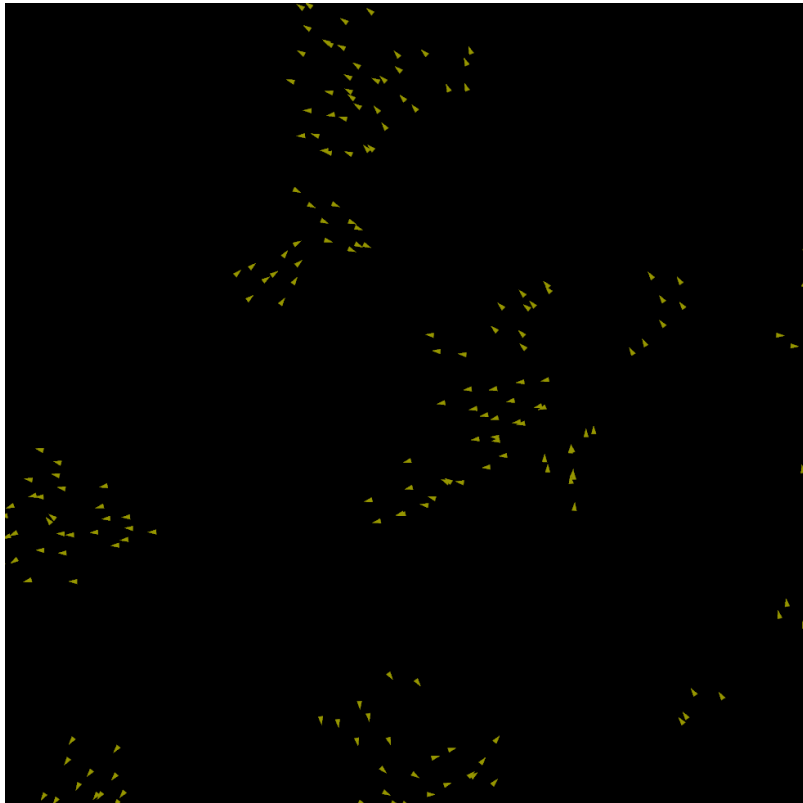
the end of the project most commits ended up being an amalgamation of whatever I was working on at the time, with changes from multiple areas of code and data files included at the same time. While I was lucky with a roll-back not being needed, had there been an issue and a commit that needed to be reversed, it would have been exponentially more work than it should have been due to the clashing nature of each commit.

## **5.6 Conclusion**

As Nim was fresh into version 1.0 when this project began, there was very limited resources or research available into the language, and at the end of the project this situation has not improved much. While Nim is a fast language with a selection of GC options far above any other language, it is nowhere close to being a usable and business-ready language, especially in the case of its JavaScript compilation. The expressiveness of its macro system and its very open approach to memory management are two of its greatest strengths but until it has time to mature and include better documentation the results of its data do not matter as its feature set is still dragging behind its competitors by a large margin.

## 6 Appendices

### 6.1 A - Boids



*Boids in action*

### 6.2 B – Boid Data

#### 6.2.1 Overview of GC modes

Position ->	Arc	Boehm	Default	Mark&Sweep	C#
AddBoids	5	2	3	4	1
DoBoids	4	3	2	1	5
Occupied Memory	5	3	2	3	
Total Time	4	3	2	1	5
Average Position	4.5	2.75	2.25	2.25	3.666667

#### 6.2.2 Comparing Nim with and without GC

ADD BOIDS		% Overhead	Do BOIDS	
Default	None		Default	None
73	66	10%	1680888	1687255
7	7		893	889
10.442	9.879		575328.9	575834.6

### 6.2.3 Comparison of time taken to add Boids to the sequence

Arc	Boehm	Default	Mark&Sweep	C#
102	84	76	79	1514
6	6	6	6	4
9.428711	8.992356	9.128	9.190311111	8.185378

### 6.2.4 Comparison of time taken to calculate Boid movements

Arc	Boehm	Default	Mark&Sweep	C#
210121	193647	172390	170352	1013719
764	763	765	762	20
58842.25	58689.93	58683.41	58621.87938	326454.4

### 6.2.5 Snapshots of GC usage during execution

	Arc	Boehm	Default	Mark&Sweep
[GC] total memory:	528384	528384	528384	528384
[GC] occupied memory:	132808	34340	120384	124928
[GC] stack scans:	33911	34340	33971	33829
[GC] stack cells:	36	37	37	36
[GC] zct capacity:	1024	1024	1024	1024
[GC] max stack size:	3192	3192	3192	3192

### 6.2.6 Comparison of Occupied Heap Memory

Arc	Boehm	Default	Mark&Sweep		
133.912	136.184	134.344	133.56		Max(MB)
105	105.064	105	105.152		Min(MB)
119.7909	119.0314	118.3328	119.1656533		Avg(MB)

### 6.2.7 Comparison of complete program times

Arc	Boehm	Default	Mark&Sweep	C#		
1.8078E+07	1.8091E+07	1.8093E+07	1.8045E+07	1.0159E+08	Max	563%
1.776E+07	1.770E+07	1.769E+07	1.766E+07	9.631E+07	Min	
1.791E+07	1.786E+07	1.786E+07	1.784E+07	9.848E+07	Avg	

## 6.3 C – Multithreading Data

Overall	C# TPL	Nim	Individual Cycles	C# TPL	Nim
Max	3953568	4108210	Max	3918240	4019534
Min	3886899	4107760	Min	94	93.8
Avg	3904691	4107846	Avg	559536	576804.4

## 6.4 D – Json For Email Client

```
"emails": [  
  {  
    "id": 0,  
    "type": "important",  
    "from": "example@mail.com",  
    "time": "10:52 AM",  
    "nickname": "James",  
    "subject": "An Example Email",  
    "message": "An example thread-starting email from James",  
    "from_id": null  
  },  
  {  
    "id": 1,  
    "type": "important",  
    "from": "me@me.com",  
    "time": "0:02 AM",  
    "nickname": "Me",  
    "subject": "An Example Email",  
    "message": "Another example message sent as a reply to message 0 from Me (does not appear in thread list)",  
    "from_id": 0  
  },  
  {  
    "id": 2,  
    "type": "important",  
    "from": "example@mail.com",  
    "time": "3:22 PM",  
    "nickname": "James",  
    "subject": "An Example Email",  
    "message": "A second reply to this thread from the original sender",  
    "from_id": 1  
  },  
  {  
    "id": 3,  
    "type": "important",  
    "from": "test@test.com",  
    "time": "9:00 AM",  
    "nickname": "Dave",  
    "subject": "An Example Email",  
    "message": "A third reply now from me, showing a full thread to completion",  
    "from_id": 2  
  },  
  {  
    "id": 4,  
    "type": "social",  
    "from": "friend@gmail.com",  
    "time": "9:45 AM",  
    "nickname": "Stacey",  
    "subject": "Friends",  
    "message": "This is an email to a friend using the social tag",  
  }  
]
```

## 6.5 E – Git Log Master Branch

✓ master	Cleanup	
	Colour boids stub	12 hours ago
	update	
	Final fixes	
	Failed attempt at Nim port	2 days ago
	Main javascript now testing	
	Beginnings of JS mail app	3 days ago
	Multithreading example complete	6 days ago
	Reorganize for new projects	
	Data collection & C# implementation	a week ago
	folder re-arrange	3 weeks ago
	data collection for Nim	
	finalized timing code, added checks for occupied memo...	
	more measuring code	a month ago
	parameterized funcs + some data collection	
	start of nim boids	
	ignore executables	
	test sprite	
	SFML c based Dll's	
	ignore ide files	
	Initial commit	

## 6.6 F – Meistertask Project Planning Log

💡 Open	🔄 In Progress	✓ Done 11	🚨 Bugs 2
<div>+</div> <div>👉</div> <div>No Tasks Drag tasks here or click + to add new tasks.</div>	<div>+</div> <div>👉</div> <div>No Tasks Drag tasks here or click + to add new tasks.</div>	<div>Basic Boids (Nim)</div> <div>Look into measuring methods</div> <div>Convert imperative code to functions for better profiler data</div> <div>collect memory data from Nim Boids</div> <div>Port Boids to c#</div> <div>Research c# multithreading abstractions</div> <div>Implement prime factors in Nim</div> <div>Implement prime factos in C#</div> <div>Record each abstraction in c'</div> <div>record MT in nim</div>	<div>Nim Boids turn too sharply</div> <div>Nim missing core DOM functionality</div> <div>+</div>

---

## 7 Bibliography

frol/completely-unscientific-benchmarks. (2019). Retrieved 21 November 2019, from <https://github.com/frol/completely-unscientific-benchmarks>

Boids. (2019). Retrieved 6 December 2019, from <https://cs.stanford.edu/people/eroberts/courses/soco/projects/2008-09/modeling-natural-systems/boids.html>

Brent-Salamin Formula -- from Wolfram MathWorld. (2019). Retrieved 6 December 2019, from <http://mathworld.wolfram.com/Brent-SalaminFormula.html>

Simple microbenchmarking in C#. (2019). Retrieved 27 November 2019, from <https://jonskeet.uk/csharp/benchmark.html>

Analyzing java memory | Dynatrace. (2019). Retrieved 27 November 2019, from <https://www.dynatrace.com/resources/ebooks/javabook/analyzing-java-memory/>

Monitoring the Activities of Garbage Collection in .NET Using CLR Profiler. (2019). Retrieved 1 December 2019, from <https://www.c-sharpcorner.com/UploadFile/7ca517/monitoring-the-activities-of-garbage-collection-in-net-usin/>

Nim's Garbage Collector. (2019). Retrieved 1 December 2019, from <https://nim-lang.org/docs/gc.html>

A guide to documenting, profiling and debugging Nim code. (2019). Retrieved 2 December 2019, from <https://nim-lang.org/blog/2017/10/02/documenting-profiling-and-debugging-nim-code.html#profiling-your-code>

Valgrind. (2019). Retrieved 2 December 2019, from <http://valgrind.org/docs/manual/ms-manual.html>

Nim Tutorial (Part III). (2019). Retrieved 6 December 2019, from <https://nim-lang.org/docs/tut3.html>

We just switched from Rust to Nim for a very large proprietary project I've been... | Hacker News. (2019). Retrieved 6 December 2019, from <https://news.ycombinator.com/item?id=9050114>

10 Languages That Compile to JavaScript — SitePoint. (2018). Retrieved 6 December 2019, from <https://www.sitepoint.com/10-languages-compile-javascript/>

Flocking Algorithm in Unity, Part 5B: Steered Cohesion Behavior. (2020). Retrieved 9 March 2020, from <https://www.youtube.com/watch?v=qzQZl09HDml&feature=youtu.be>



ICS 311 #22:Multithreaded Algorithms . (2020). Retrieved 21 May 2020, from <http://www2.hawaii.edu/~suthers/courses/ics311f17/Notes/Topic-22.html>

---

## 8 References

- <sup>i</sup> Rumpf, A. *Version 1.0 released*. (2020). Retrieved 18 May 2020, from <https://nim-lang.org/blog/2019/09/23/version-100-released.html>
- <sup>ii</sup> Nim Programming Language. (2020). Retrieved 18 May 2020, from <https://nim-lang.org/>
- <sup>iii</sup> Differences between Stack and Heap. (2020). Retrieved 20 May 2020, from <http://net-informations.com/faq/net/stack-heap.htm>
- <sup>iv</sup> Araujo J, Matos R, Maciel P. Experimental Evaluation of Software Aging Effects on the Eucalyptus Cloud Computing Infrastructure. Retrieved 18 May 2020, from <https://dl.acm.org/doi/pdf/10.1145/2090181.2090185>
- <sup>v</sup> Wilson, P. Uniprocessor garbage collection techniques. *Memory Management*, 1-42. doi: 10.1007/bfb0017182
- <sup>vi</sup> Nim's Memory Management. (2020). Retrieved 18 May 2020, from <https://nim-lang.org/docs/gc.html>
- <sup>vii</sup> Rumpf, A. Nim's Garbage Collector. (2020). Retrieved 20 May 2020, from <https://nim-lang.org/0.17.2/gc.html>
- <sup>viii</sup> Valgrind: Tool Suite. (2020). Retrieved 20 May 2020, from <https://valgrind.org/info/tools.html>
- <sup>ix</sup> Barney. B. POSIX Threads Programming. (2020). Retrieved 20 May 2020, from <https://computing.llnl.gov/tutorials/pthreads/>
- <sup>x</sup> threads. (2020). Retrieved 20 May 2020, from <https://nim-lang.org/docs/threads.html>
- <sup>xi</sup> Davies, A. Async in C# 5.0. (2020). Retrieved 20 May 2020, from [https://books.google.co.uk/books?hl=en&lr=&id=1On1glEbTfIC&oi=fnd&pg=PR3&dq=c%23+task&ots=Sg3ULijHS8&sig=s3Eu-Byd9AnkU--zrc36WxqekJ4&redir\\_esc=y#v=onepage&q=c%23%20task&f=false](https://books.google.co.uk/books?hl=en&lr=&id=1On1glEbTfIC&oi=fnd&pg=PR3&dq=c%23+task&ots=Sg3ULijHS8&sig=s3Eu-Byd9AnkU--zrc36WxqekJ4&redir_esc=y#v=onepage&q=c%23%20task&f=false)
- <sup>xii</sup> Calvert C, Kulkarni, D. Essential LINQ. (2020). Retrieved 20 May 2020, from [https://books.google.co.uk/books?hl=en&lr=&id=PoYFLgfkELsC&oi=fnd&pg=PR11&dq=LINQ+c&ots=Okwya2qtXV&sig=VDudbPpuFOVDbk-yla00DMc2ScU&redir\\_esc=y#v=onepage&q=LINQ%20c&f=false](https://books.google.co.uk/books?hl=en&lr=&id=PoYFLgfkELsC&oi=fnd&pg=PR11&dq=LINQ+c&ots=Okwya2qtXV&sig=VDudbPpuFOVDbk-yla00DMc2ScU&redir_esc=y#v=onepage&q=LINQ%20c&f=false)
- <sup>xiii</sup> Kanjilal, J. (2020). How to work with Parallel LINQ in C#. Retrieved 20 May 2020, from <https://www.infoworld.com/article/3021870/how-to-work-with-parallel-linq-in-c.html>
- <sup>xiv</sup> frol/completely-unscientific-benchmarks. (2020). Retrieved 20 May 2020, from <https://github.com/frol/completely-unscientific-benchmarks>
- <sup>xv</sup> Treap - Competitive Programming Algorithms. (2020). Retrieved 20 May 2020, from [https://cp-algorithms.com/data\\_structures/treap.html](https://cp-algorithms.com/data_structures/treap.html)
- <sup>xvi</sup> Robie J, What is the Document Object Model?. (2020). Retrieved 20 May 2020, from <https://www.w3.org/TR/WD-DOM/introduction.html>
- <sup>xvii</sup> V8 JavaScript engine. (2020). Retrieved 20 May 2020, from <https://v8.dev/>
- <sup>xviii</sup> Using template engines with Express. (2020). Retrieved 20 May 2020, from <https://expressjs.com/en/guide/using-template-engines.html>
- <sup>xix</sup> ASP.NET MVC Architecture. (2020). Retrieved 20 May 2020, from <https://www.tutorialsteacher.com/mvc/mvc-architecture>
- <sup>xx</sup> Dhaduk, H., Kasundra, P., Dhaduk, H., & Kaneriya, T. (2019). Node.JS Use Case: When & How Node.JS Should be Used | Simform. Retrieved 20 May 2020, from <https://www.simform.com/nodejs-use-case/#Node.js-uses>
- <sup>xxi</sup> Reynolds, C (1987). Flocks, Herds, and Schools: A Distributed Behavioural Model
- <sup>xxii</sup> Rosetta Code. (2020). Retrieved 20 May 2020, from [https://www.rosettacode.org/wiki/Parallel\\_calculations#C.23](https://www.rosettacode.org/wiki/Parallel_calculations#C.23)
- <sup>xxiii</sup> threadpool. (2020). Retrieved 21 May 2020, from <https://nim-lang.org/docs/threadpool.html>
- <sup>xxiv</sup> Bulma Overview. (2020). Retrieved 21 May 2020, from <https://bulma.io/documentation/overview/>
- <sup>xxv</sup> strformat. (2020). Retrieved 21 May 2020, from <https://nim-lang.org/docs/strformat.html>