Towards a Dynamic Pipeline Framework implemented in (parallel) Haskell*

Juan Pablo Royo Sales[a], Edelmira Pasarella[a], Cristina Zoltan[a], Maria-Esther Vidal[b]

[a] *Universitat Politecnica de Catalunya, 08034, Barcelona, Spain*

[b] *TIB/L3S Research Centre at the University of Hannover, Hannover, Germany*

Abstract

Streaming processing has given rise to new computation paradigms to provide effective and efficient data stream processing. The most important features of these new paradigms are the exploitation of parallelism, the capacity to adapt execution schedulers, reconfigure computational structures, adjust the use of resources according to the characteristics of the input stream and produce incremental results. The Dynamic Pipeline Paradigm (DPP) is a naturally functional approach to deal with stream processing. This fact encourages us to use a purely functional programming language for DPP. In this work, we tackle the problem of assessing the suitability of using (parallel) Haskell to implement a Dynamic Pipeline Framework (DPF). The justification of this choice is twofold. From a formal point of view, Haskell has solid theoretical foundations, providing the possibility of manipulating computations as primary entities. From a practical perspective, it has a robust set of tools for writing multithreading and parallel computations with optimal performance. As proof of concept, we present an implementation of a dynamic pipeline to compute the weakly connected components of a graph (WCC) in Haskell (a.k.a. $\mathsf{DP}_{\mathsf{WCC}}$). The $\mathsf{DP}_{\mathsf{WCC}}$ behavior is empirically evaluated and compared with a solution provided by a Haskell library. The evaluation is assessed in three networks of different sizes and topology. Performance is measured in terms of the time of the first result, continuous generation of results, total time, and consumed memory. The results suggest that $\mathsf{DP}_{\mathsf{WCC}}$, even naive, is competitive with the baseline solution available in a Haskell library. $\mathsf{DP}_{\mathsf{WCC}}$ exhibits a higher continuous behavior and can produce the first result faster than the baseline. Albeit initial, these results put in perspective the suitability of Haskell's abstractions for the implementation of DPF. Built on them, we will develop a general and parametric DPF in the future.

Keywords: Dynamic pipeline, Haskell, parallelism, concurrency

1. Introduction

Effective streaming processing of large amounts of data has been studied for several years [2, 3, 5] as a key factor providing fast and incremental results in big data algorithmic problems. One of the most explored techniques, regardless of the approach, is the exploitation of parallel techniques to take advantage of the available computational power as much as possible. In that regard, the Dynamic

*This work has been partially supported by MINECO and FEDER funds under grant TIN2017-86727-C2-1-R.

Pipeline Paradigm (DPP) [14] has lately emerged as one of the models that exploit data streaming processing using a dynamic pipeline parallelism approach [5]. This computational model has been designed with a functional focus, where the main components of the paradigm are functional stages or pipes which dynamically enlarge and shrink depending on incoming data.

One of the biggest challenges of implementing a Dynamic Pipeline Framework (DPF) is to find a proper set of tools and programming language which can take advantage of both of its primary aspects: i) *fast parallel* processing and ii) *strong theoretical* foundations that manage computations as first-class citizens. Haskell Programming Language (Haskell) is a statically typed pure functional language which has been designed and evolved from its birth in 1987, on strong theoretical foundations where computations are primary entities, and at the same time has been providing a powerful set of tools for writing multithreading and parallel programs with optimal performance [7, 8].

Problem Research and Objective: The main objective of this work is to explore the feasibility of using a Functional Programming (FP) language to implement a DPF. In particular, we tackle the problem of establishing the basis of an implementation of a DPF in Haskell, a pure functional language. This is, our aim is to determine the particular features (i.e., versions and libraries) of this language that will allow for an efficient implementation of the DPF. To be concrete, through a particular and very relevant problem as the computation of the Weak Connected Components (WCC) of a graph, we study the critical features required in Haskell for a DPF implementation.

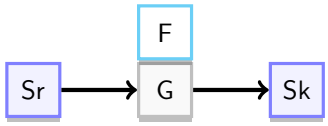
Approach: In Section 3 we define a first approach for DPP Framework in Haskell. In section 4 we present an algorithm for enumerating WCC using DPP and its implementation using Haskell. Finally, sections 5 and 6 report a set of experiments conducted to support our hypothesis, as well as the analysis of the results obtained from those experiments.

Contributions: A proof of concept of the implementation of a DP for WCC using Haskell; the results of the empirical study suggests that Haskell is a suitable language for implementing DPP. This work also contributes to building the first abstraction approximation of a future framework/library of this computational model in that language.

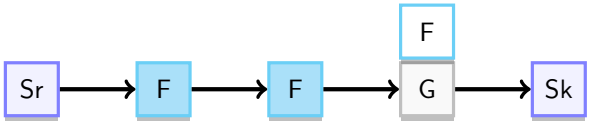
2. Dynamic Pipeline Paradigm

The *Dynamic Pipeline Paradigm* (DPP) [14] is a data-driven computational model based on a one-dimensional and unidirectional chain of stages connected by means of channels synchronized by data availability. This chain of stages is a computational structure called *Dynamic Pipeline* (DP). A DP stretches and shrinks depending on the spawning and the lifetime of its stages, respectively. Modeling an algorithmic solution as a DP corresponds to define a dynamic computational structure in terms of four kinds of stages: *Source* (Sr), *Generator* (G), *Sink* (Sk) and *Filter* (F) stages. To be concrete, the specific behaviour of each stage to solve a particular problem must be defined as well as the number and the type of the I/O channels connecting them. Channels are unidirectional according to the flow of the data. The *Generator* stage is in charge of spawning *Filter* stage instances. This particular behavior of the *Generator* gives the elastic capacity to DPs. *Filter* stage instances are stateful operators in the sense described in [12]. This is, *Filter* instances have a state. The deployment of a DP consists in setting up the initial configuration depicted in Figure 1a. The activation of a DP starts when a stream of data items arrives to the initial configuration of the DP. In particular, when a stream data items arrives to the *Source* stage. During the execution, the *Generator* stage spawns *Filter* stage instances according to incoming data and the *Generator* defined behavior. This evolution is illustrated in Figure 1b. If the stream data is bounded, the computation finish when the

lifetime of all the stages of the active DP have finished. Otherwise, if the stream data is unbounded, the DP remains active and incremental results are output.



(a) Initial configuration of a Dynamic Pipeline. An initial DP consists of three stages: Sr, G together its filter parameter F, and Sk. These stages are connected through its channels –represented by right arrows– as shown in this figure.



(b) Evolution of a DP. After creating some filter instances (shadow Filter squares) of the filter parameter (light Filter square) in the Generator, the DP has stretched.

Figure 1: Dynamic Pipeline configuration

A *Dynamic Pipeline Framework* (DPF) is a software system that allows to implement and activate their DPs. In particular, a DPF provides a way to specify dynamic pipelines and an interpreter to run triggered DPs that ensures, among others, proper resource utilization, safeness, and failure recovery. DPF is implemented in some Host Language (HL)[1] and comprises three main modules: i) Domain-specific Language (DSL), which is representing DP in an Embedded Domain-specific Language (EDSL) to allow defining the different components of a DP as well as their connections, ii) Interpreter of DSL (IDL) which has all the functions needed to set up a DP, and iii) Runtime System (RS) to trigger and execute a DP. Figure 2 depicts the DPF architecture.

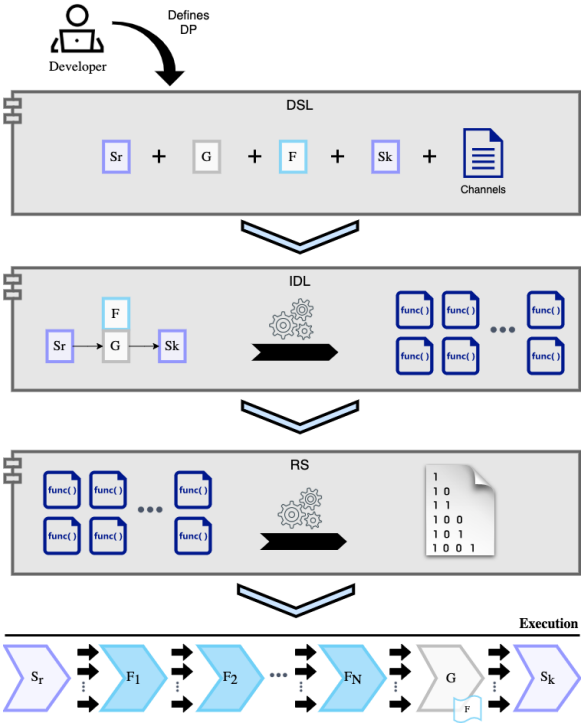


Figure 2: Dynamic Pipeline Framework Architecture

As we can appreciate in Figure 2, a user with DP and HL knowledge (called *Developer* here), interacts with the DSL module. Through this module, the user defines the behaviors and connections of the four stages of a DP, i.e., the channels between stages and their types, input data, and the *Filter* template. Indeed, IDL sets up the initial configuration as described in Figure 1a and, interprets and translates user definitions to specific HL constructs and implementation. This means that the user needs to write the algorithms that each stage should perform, interacting with the available channels declared for those stages. Considering a particular DP, the RS module contains an entry point function to allow the user to run it. The execution of the DP takes place as depicted in Figure 1b.

[1] A Host Language means the Programming Language that is used to implement DPF, i.e. Haskell, Go, Rust, etc.

The DPF requirements are defined based on stream processing systems described by Röger and Mayer [12]. A DP spans a network of stages, hence to guarantee a good performance, it is needed to ensure low latency and high throughput when a DP is activated. To achieve this goal, it is necessary a high level of parallelization. Besides, to increase the parallelization level, it is required a mechanism that supports the deployment of the DP stateful stages in a multi-core server. In particular, this mechanism must support the DP elasticity, i.e., the spawn/kill of filter instances. Additionally, the realization of channels must be supported and in-order computation ensured.

3. A Dynamic Pipeline Framework: First Approach

In this section, we focus on the design and implementation of IDL and RS in parallel Haskell. IDL and RS can be considered the core of the DPF since, according to the architecture shown in Figure 2, these components of the DPF are in charge of setting up and executing a DP.

IDL implementation requires certain characteristics that chosen language and tools should provide to reach the desired objective. Regarding this, and according to what has been described on section 2, we detail the tools and mechanisms selected for each component of a DP[2].

DP *Monad.* A functional programming language like Haskell, it is a proper host for representing those dynamic computations, not only due to its FP nature, but also for it is strong type system which ensure safeness on the data representation pipeline and the computational sequence. Therefore, there is no need to select any other additional library rather than using *Monads* [13] to represent the chain of computations and the dependency between those computations in the implementation of a pipeline. In this sense, representing a DP as a *Monad*, *Monad Laws* [11] are ensured. In particular, the *Monad* associativity law, i.e., $((m >>= f) >>= g = m >>= (\lambda x.fx >>= g))$, guarantees the execution flows of Sr, G and Sk.

Filter / Stage. Abstractions like *Recursive Schemes* [9] has been deeply explored and used in Haskell. They allow for the representation, in a formal and simple way, of dynamic functional structures like *anamorphisms* and *catamorphisms* [9]. This abstraction is already provided in the base library in `fold` and `unfold` combinators. In this first solution, we have chosen to implement these simplest abstractions. However, we hypothesize if more *Recursive Scheme* combinators are implemented, the proposed approach can take advantage of other properties and laws, e.g., the ones supported by *algebra and coalgebra* structures [9].

Parallelization. One of the most important components of the implementation is the selection of a concurrency library to support an intensive parallelization workload. Parallelization techniques and tools have been intensively studied and implemented in Haskell [8]. Indeed, it is well known that green or light threads and spark allow for spawning thousands to millions of parallel computations. These parallel computations do not penalize performance when compare with Operative System (OS) level threading [7]. A straightforward assumption to achieve this could be to use `monad-par` library [26, 8]. Nevertheless, in this experimental work, we have discarded the use of sparks [32] because we can achieve the level of required parallelism spawning green threads only. This is caused by the nature of DPP, where pipeline parallelism and not data parallelism is a structural processing mechanism. The next obvious choice is to use `forkIO :: IO () -> IO ThreadId` from `base` library [24]. However, that would imply handling all the threads lifecycle, terminations, and

[2] All the Source Code of this work can be found in this Github Repository

errors programmatically without major combinators or abstractions to deal with them. Therefore, we choose `async` library [15] which enables to spawn asynchronous computations [7] on Haskell and at the same time, it provides useful combinators to managing thread terminations and errors.

Channels. We have several techniques to our disposal to communicate between threads or sparks in Haskell like `MVar` or concurrent safe mechanisms like Software Transactional Memory (STM) [4]. Moreover, we have at our disposal *Channels* abstractions based on both mentioned communication techniques. In that sense, for conducting the communication between dynamic stages and data flowing in a DP, we have selected `unagi-chan` library [37] which provides the following advantages to our solution: Firstly, `MVar` channel without using STM. This allows avoiding internal locking for concurrent access. In this case, we can use this advantage because in a DP, one specific stage which is running in a separated thread, can only access to its *I/O* channels for reading/writing accordingly and those operations are not concurrently shared by other threads (stages) for the same channels. Second, non-blocking channels. `unagi-chan` library contains blocking and non-blocking channels for reading. This aspect is key to gain speed up on the implementation. Third, the library is optimized for $x86$ architectures with use of low-level `fetch-and-add` instructions. Finally, `unagi-chan` is $100x$ faster on Benchmarking compare with STM and default base `Chan` implementations.

```
1  runParallelDP :: Handle -> IO ()
2  runParallelDP h = source h >>= generator >>= sink
3
4  source :: Handle -> IO (DP.Stream MySource MySink)
5  source h = fromFile h >>= (|>> parseSource)
6
7  sink :: MySink -> IO ()
8  sink = DP.mapM (R.putStrLn . show)
9
10 fromFile :: Handle -> IO (DP.Stream ByteString MySource)
11 fromFile h = DP.unfoldM (B.hGetLine h) (R.hIsEOF h)
```

Listing 3.1: An overview of a generic DP source code in Haskell

As we said in section 2, to implement a DPF the development tool must provide a high level of parallelization and a mechanism that supports the deployment of DP*s* in multi-core servers allowing the elasticity and channels. As explained above, Haskell meets these requirements. In listing 3.1 we can see a source code corresponding to the implementation of a generic DP in Haskell.

4. Computing (Weak) Connected Components

Let us consider the problem of computing the (weak) connected components of a graph G using DPP. A connected component of a graph is a subgraph in which any two vertices are connected by paths. Thus, finding connected components of a directed graph implies obtaining the minimal partition of the set of nodes induced by the relationship *connected*, i.e., there is a path between each pair of nodes. The input of the Dynamic Pipeline for computing the WCC of a graph, $\mathsf{DP}_{\mathsf{WCC}}$, is

a sequence of edges ending with eof[3]. The connected components are output as soon as they are computed, i.e., they are produced incrementally.

4.1. DP_{WCC} Definition

$\mathsf{DP_{WCC}}$ is defined in terms of the behavior of its four kinds stages: *Source* ($\mathsf{Sr_{WCC}}$), *Generator* ($\mathsf{G_{WCC}}$), *Sink* ($\mathsf{Sk_{WCC}}$), and *Filter*($\mathsf{F_{WCC}}$) stages. Additionally, the channels connecting these stages must be defined. In $\mathsf{DP_{WCC}}$, stages are connected linearly and unidirectionally through the channels $\mathsf{IC_E}$ and $\mathsf{IC_{set(V)}}$. Channel $\mathsf{IC_E}$ carries edges while channel $\mathsf{IC_{set(V)}}$ conveys sets of connected vertices. Both channels end by the eof mark. The initial configuration of $\mathsf{DP_{WCC}}$ is $\mathsf{Sr_{WCC}} \rightrightarrows \mathsf{G_{WCC}} \rightarrow \mathsf{Sk_{WCC}}$, where $\rightrightarrows$ represents channels $\mathsf{IC_E}$ and $\mathsf{IC_{set(V)}}$ while $\rightarrow$ represents the channel $\mathsf{IC_{set(V)}}$.

Once activated the initial $\mathsf{DP_{WCC}}$, the stream of edges is fed into $\mathsf{Sr_{WCC}}$ and $\mathsf{Sk_{WCC}}$ produces the resulting connected components. $\mathsf{G_{WCC}}$ has as parameter the template of the stage $\mathsf{F_{WCC}}$. When an edge (v, w) arrives to $\mathsf{G_{WCC}}$, it spawns a new instance of $\mathsf{F_{WCC}}$ before $\mathsf{G_{WCC}}$. For example, the first time a filter instance is spawned, the $\mathsf{DP_{WCC}}$ evolves to this one: $\mathsf{Sr_{WCC}} \rightrightarrows \mathsf{F_{WCC}} \rightrightarrows \mathsf{G_{WCC}} \rightarrow \mathsf{Sk_{WCC}}$. The state of this new filter instance is initialized with the vertices $\{v, w\}$. When eof arrives to $\mathsf{G_{WCC}}$, it connects previous filter instance to $\mathsf{Sk_{WCC}}$ through $\mathsf{IC_{set(V)}}$; then, $\mathsf{G_{WCC}}$ dies and the $\mathsf{DP_{WCC}}$ evolves as follows: $\mathsf{Sr_{WCC}} \rightrightarrows \mathsf{F_{WCC}} \rightrightarrows \cdots \mathsf{F_{WCC}} \rightrightarrows \mathsf{Sk_{WCC}}$. The behavior of $\mathsf{F_{WCC}}$ is given by a sequence of two actors (scripts). In what follows we denote these actors by $\mathsf{actor_1}$ and $\mathsf{actor_2}$, respectively. The script $\mathsf{actor_1}$ keeps a set of connected vertices (CV) in the state of the $\mathsf{F_{WCC}}$ instance. When an edge e arrives, if an endpoint of e is present in the state, then the other endpoint of e added to CV. Edges without incident endpoints are passed to the next stage. When eof arrives at channel $\mathsf{IC_E}$, it is passed to the next stage and the script $\mathsf{actor_2}$ starts its execution. If script $\mathsf{actor_2}$ receives a set of connected vertices CV in $\mathsf{IC_{set(V)}}$, it determines if the intersection between CV and the nodes in its state is not empty. If so, it adds the nodes in CV to its state. Otherwise, the CV is passed to the next stage. Whenever eof is received, $\mathsf{actor_2}$ passes–through $\mathsf{IC_{set(V)}}$– the set of vertices in its state and the eof mark to the next stage; then, it dies. The behavior of $\mathsf{Sr_{WCC}}$ corresponds to the identity transformation over the data stream of edges. As edges arrive, they are passed through $\mathsf{IC_E}$ to the next stage. When receiving eof on $\mathsf{IC_E}$, this mark is put on both channels. Then, $\mathsf{Sr_{WCC}}$ dies.

4.2. $\mathsf{DP_{WCC}}$ *Implementation*

As we said before, the $\mathsf{DP_{WCC}}$ implementation has been made as a proof of concept to understand and explore the limitations and challenges that we could find in the development of a future DPF in Haskell. In Section 3 we emphasize that the focus of DPF in Haskell is on the IDL component. Hence, the development of the $\mathsf{DP_{WCC}}$ is as general as possible using most of the constructs and abstractions require by the IDL.

As we have seen on section 3 in listing 3.1, all the Stages *Source* ($\mathsf{Sr_{WCC}}$), *Generator* ($\mathsf{G_{WCC}}$) and *Sink* ($\mathsf{Sk_{WCC}}$) are represented as monadic computations composed in `runParallelDP` function. As we can appreciate, `runParallelDP` is running in the context of a `Handle` descriptor which is the stream input file that is feeding $\mathsf{Sr_{WCC}}$. That input is injected incrementally in $\mathsf{IC_E}$ as we can see in this *anamorphism* `unfoldM` with the help of the lazy `ByteString` reader. As long as a new edge (v, w) is received by `generator` from $\mathsf{IC_E}$, a new *Filter*($\mathsf{F_{WCC}}$) (monadic computation) is interposed in the execution chain. This can be seen with the use of `foldrS` here. Channel connection and disconnection are implicit by function parameters and recursion since the new $\mathsf{F_{WCC}}$ is receiving as actual parameters, $\mathsf{IC_E}$, and $\mathsf{IC_{set(V)}}$ belonging to $\mathsf{G_{WCC}}$. In the next execution loop, $\mathsf{G_{WCC}}$ is pulling

[3]Note that there are neither isolated vertices nor loops in the source graph G.

edges from $\mathsf{IC_E}$ provided by previously $\mathsf{F_{WCC}}$ created instance. It is important to remark that all these monadic computations are spawned in parallel through the use of `async` combinator [15]. That means different threads continue reading and writing from channels without blocking progress if data is available. Actors (scripts) are described as computation as well as the rest of the stages but, in this particular case, they are not spawned in different threads because they must run sequentially in the same $\mathsf{F_{WCC}}$ thread according to the definition. Script $\mathsf{actor_1}$ and $\mathsf{actor_2}$ are represented by functions `actor1` and `actor2` respectively. $\mathsf{actor_1}$ is reading all the edges (v, w) pulled from $\mathsf{IC_E}$ and keeping in the filter state a list with all the vertices that are pairwise connected (see repository for more details), If the edge is not connecting to any vertex of the list it passes through the next computational stage as can be seen here. In all channels eof is represented using `Maybe` type, meaning that when someone pushes a `Nothing` value to the channel the next reader can detect eof. This is why we are folding the channel with `maybe` combinator as it can be appreciated here. Once $\mathsf{actor_1}$ finishes $\mathsf{actor_2}$ starts its computation as it is described in these lines. In this case $\mathsf{actor_2}$ is reading from $\mathsf{IC_{set(V)}}$ channel that could contain the previous calculated `ConnectedComp`. Using `union` and `intersect` combinators allows to merge the calculated list of vertices that are connected with previously calculated connected components by other $\mathsf{F_{WCC}}$. In this implementation `ConnectedComp` is a `newtype` over `IntSet` to speed up computation for intersection and union [19]. Moreover, `doActor` which is an inner function of `actor2`, is pushing to $\mathsf{IC_{set(V)}}$ channel all `ConnectedComp` as they are treated. $\mathsf{G_{WCC}}$ computation passes to $\mathsf{Sk_{WCC}}$ computation the reference to $\mathsf{IC_{set(V)}}$ channel which is going to pull `ConnectedComp` as long as it is available, and printing that information in the standard output incrementally. Once $\mathsf{Sk_{WCC}}$ receives a `Nothing` (eof) value from $\mathsf{IC_{set(V)}}$ the whole computation ends.

5. Empirical Evaluation

The empirical study aims at evaluating the performance of $\mathsf{DP_{WCC}}$ when implemented in Haskell. Our goal is to answer the following research questions:

RQ1) Does $\mathsf{DP_{WCC}}$ in Haskell support the dynamic parallelization level that $\mathsf{DP_{WCC}}$ requires? **RQ2)** Is $\mathsf{DP_{WCC}}$ in Haskell competitive compared with default implementations on base libraries for the same problem? **RQ3)** Does $\mathsf{DP_{WCC}}$ in Haskell handle memory efficiently?

We have conducted different kinds of experiments to test our assumptions and verify the correctness of the implementation. First, we have performed an *Implementation Analysis* in which we have selected some graphs from Stanford Network Data Set Collection (SNAP) [34] and analyze how the implementation behaves under real-world graphs if it timeouts or not and if it is producing correct results in terms of the amount of WCC that we know beforehand. We have also tested the implementation doing a *Benchmark Analysis* where we focus on two different types of benchmarks. On the one hand, using `criterion` library [20], we have evaluated a benchmark between our solution and WCC algorithm implemented in `containers` Haskell library [19] using `Data.Graph`. On the other hand, we have compared if the results are being generated incrementally in both cases and how that is done during the pipeline execution time. This last analysis has been conducted using `diefpy` tool [1, 21]. Finally, we have executed a *Performance Analysis* in which we have to gather profiling data from Glasgow Haskell Compiler (GHC) for one of the real-world graphs, to measure how the program performs regarding multithreading and memory allocation.

5.1. Running Architecture

All the experiments have been executed in a $x86$ 64 bits architecture with a 6-*Core Intel Core i7* processor of $2,2$ GHz which can emulate up to 12 virtual cores. This processor has *hyper-threading*

enable. Regarding memory, the machine has $32GB$ *DDR4* of RAM, 256 KB of L2 cache memory, and 9 MB of L3 cache.

5.2. Haskell Setup

Regarding specific libraries and compilations flags used on Haskell, we have used GHC version 8.10.4. We have also used the following set of libraries: `bytestring 0.10.12.0` [17], `containers 0.6.2.1` [19], `relude 1.0.0.1` [31] and `unagi-chan 0.4.1.3` [37]. The use of `relude` library is because we disabled `Prelude` from the project with the language extension `NoImplicitPrelude` [23]. Regarding compilation flags (GHC options) we have compiled our program with `-threaded`, `-O3`, `-rtsopts`, `-with-rtsopts=-N`. Since we have used `stack` version 2.5.1 [33] as a building tool on top of GHC the compilation command is `stack build`[4].

5.3. DataSets

For all the experiments, we have used the following networks taken from SNAP [34]. In this particular experiment setup, we have selected the following specific data sets that can be found here [28, 27, 29]

Network	**Nodes**	**Edges**	**Diameter**	**#WCC**	**#Nodes Largest WCC**
Enron Emails	36692	183831	11	1065	33696 (0.918)
Astro Physics Collaboration Net	18772	198110	14	290	17903 (0.954)
Google Web Graph	875713	5105039	21	2746	855802 (0.977)

Table 1: DataSet of Graphs Selected

The criteria for selecting the networks have been followed the idea or testing the solution in more complex graphs, in which all of them are undirected but with different sizes concerning its number of nodes as we can see in Table 1.

5.4. Experiments Definition

E1: Implementation Analysis. In this experiment, we measure GHC statistics running time enabling `+RTS -s` flags. The metrics that we measure are *MUT Time* which is the amount of time in seconds GHC is running computations and *GC Time* which is the number of seconds that GHC is running garbage collector. *Total execution time* is the sum of both in seconds. At the same time, we are going to check the correctness of the output counting the number of WCC generated by the algorithm against the already known topology of it in subsection 5.3. The experiment's primary goal is to help answer the research question [RQ2].

E2: Benchmark Analysis. In this experiment, we conduct two benchmark analysis over execution time comparing DP-Haskell with Haskell `containers` default implementation. In the first benchmark analysis, we use `criterion` [20] tool in Haskell which runs over four iterations of each of the algorithms to get a mean execution time in seconds and compare the results in a plot. In the second benchmark, we use Diefficency Metrics (Dm) Tool *diefpy* [21] in order to measure with the ability of DPP model to generate results incrementally [1]. This is one of the strongest feature of

[4]For more information about package.yaml or cabal file please check https://github.com/jproyo/upc-miri-tfm/tree/main/connected-comp

DPP Paradigm since it allows process and generate results without no need of waiting for processing until the last element of the data source. This kind of aspect is essential not only for big data inputs where perhaps the requirements allow for processing until some point of the time having partial results but at the same time is important to process unbounded streams. The experiment's primary goal is to help answer the research question [RQ2] as well.

E3: Performance Analysis. In this experiment, we measure internal parallelism in GHC and memory usage during the execution of one of the example networks. The motivation of this is to verify empirically how DP-Haskell is handling parallelization and memory usage. This experiment is conducted using two tools, *ThreadScope* [36] for conducting multithreading analysis and *eventlog2html* [22] to conduct memory usage analysis. Regarding multithreading analysis the metrics that we measure are the distribution of threads among processors over execution time which is how many processors are executing running threads over the whole execution; and the mean number of running threads per time slot which is calculated by zooming in 8 time slots and taking the mean number of threads per processor to see if it is equally distributed among them. In regards to memory management, the metric that we measure is the amount of memory in MB consumed per data type during the whole execution time. The experiment helps to answer the research questions [RQ1,RQ3].

6. Discussion of Observed Results

6.1. Experiment: E1

The following represents the execution for running these graphs on our DPP implementation.

Network	Exec Param	MUT Time	GC Time	Total Time
Enron Emails	`+RTS -N4 -s`	2.797s	0.942s	3.746s
Astro Physics Coll Net	`+RTS -N4 -s`	2.607s	1.392s	4.014s
Google Web Graph	`+RTS -N8 -s`	137.127s	218.913s	**356.058s**

Table 2: Execution times

It is important to point out that since the first two networks are smaller in the number of edges compared with *web-Google*, executing those with 8 cores as the `-N` parameters indicates, does not affect the final speed-up since GHC is not distributing threads on extra cores because it handles the load with 4 cores only.

As we can see in Table 2, we are obtaining remarkable execution times for the first two graphs and it seems not to be the case for *web-Google*. Doing a deeper analysis on the topology of this last graph, we can see according to Table 1 that the number of *Nodes in the largest WCC* is the highest one. This means that there is a WCC which contains 97.7% of the nodes. Moreover, we can confirm that if we analyze even deeper how is the structure of that WCC with the output of the algorithm, we can notice that the largest WCC is the last one on being processed. Having that into consideration we can state that due to the nature of our algorithm which needs to wait for collecting all the vertices in the `actor2` filter stage it penalizes our execution time for that particular case. A more elaborated technique for implementing the actors is required to speed up execution.

Regarding the correctness of the output, we have verified with the outputs that the number of connected components is the same as the metrics already gathered in Table 1.

6.2. Experiment: E2

Criterion Benchmark. In Figure 3, orange bars report the time taken by `Data.Graph` in Haskell `containers` library [19]. Blue light bars represent the time taken by DP-Haskell.

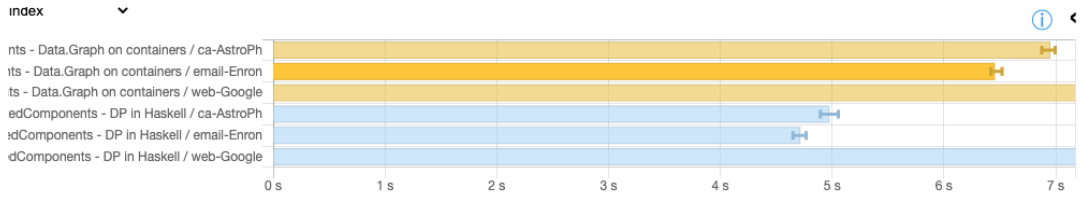


Figure 3: Benchmark 1 - DP in Haskell vs. Data.Graph Haskell

Figure 3 shows that DP-Haskell solution is 1.3 faster compare with Haskell `containers` library. Despite this, if we zoom in Figure 3, it can be observed that DP-Haskell solution is slower compared with Haskell `containers`; the reasons behind this have been explained in subsection 6.1.

Regarding mean execution times for each implementation on each case measure by `criterion` library [20], we can display the following results:

Network	**DP-Haskell**	**Haskell `containers`**	**Speed-up**
Enron Emails	4.68s	6.46s	1.38
Astro Physics Coll Net	4.98s	6.95s	1.39
Google Web Graph	386s	106s	-3.64

Table 3: Mean Execution times

These results allow for answering Question [Q2]. We already had a partial answer with the previous experiment E1 about [Q2] (section 5) where we have seen that the graph topology is affecting the performance and the parallelization, penalizing DP-Haskell for this particular case. In this benchmark, the solution against a non-parallel `containers` `Data.Graph` confirms the hypothesis.

Diefficency Metrics. Some considerations are needed before starting to analyze the data gathered with Dm tool. Firstly, the tool is plotting the results according to the traces generated by the implementation, both DP-Haskell and Haskell *containers*. By the nature of DPP model, we can gather or register that timestamps as long as the model is generating results. In the case of Haskell `containers`, this is not possible since it calculates WCC at once. This is not an issue and we still can check at what point in time all WCC in Haskell `containers` are generated. In those cases, we are going to see a straight vertical line.

It is important to remark that we needed to scale the timestamps because we have taken the time in nanoseconds. After all, the incremental generation between one WCC and the other is very small but significant enough to be taken into consideration. Thus, if we left the time scale in integer milliseconds, microseconds, or nanoseconds integer part it cannot be appreciated. In case of escalation, we are discounting the nanosecond integer of the first generated results resulting in a time scale that starts close to 0. This does not mean that the first result is generated at 0 time, but we are discarding the previous time to focus on how the results are incrementally generated.

Having said that, we can see the results of Dm which are presented in two types of plots. The first one is regular line graphs in where the x axis shows the time escalated when the result was

generated and the y axis is showing the component number that was generated at that time. The second type of plot is a radar plot in which shows how the solution is behaving on the dimensions of Time for the first tuple (TFFT), Execution Time (ET), Throughput (T), Completeness (Comp) and Diefficiency first t time units (dief@t) and how are the tension between them; all these metrics are higher is better. All the details about these metrics are explained here [1].

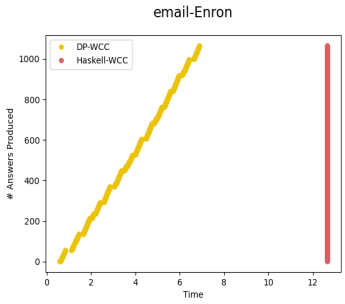


Figure 4: email-Enron Dm

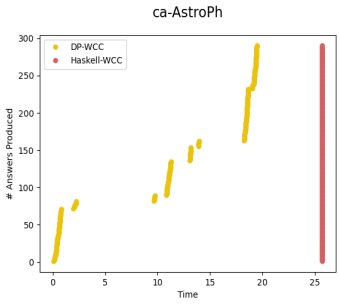


Figure 5: ca-AstroPh Dm

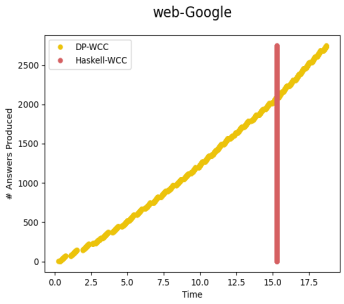


Figure 6: web-Google Dm

Based on the results shown in all the figures above, all the solutions in DP-Haskell are being generated incrementally, but there is some difference that we would like to remark. In the case of *email-Enron* and *ca-AstroPh* graphs as we can see in Figure 4 and Figure 5, there seems to be a more incremental generation of results. This is behavior is measured with the values of Diefficiency first t time units (dief@t). *ca-AstroPh* as it can be seen in Figure 5, is even more incremental showing a clear separation between some results and others. The *web-Google* network which is shown in Figure 6, is a little more linear and that is because all the results are being generated with very little difference in time between them. This is due to the fact of the explained reasons in subsection 6.1. Having the biggest WCC at the end of *web-Google* DPP algorithm it is retaining results until the biggest WCC can be solved, which takes longer.

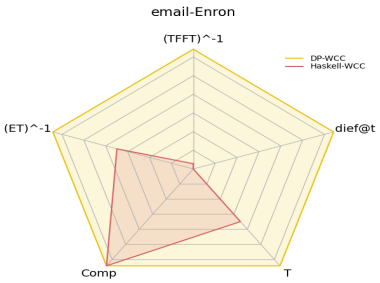


Figure 7: email-Enron Dm

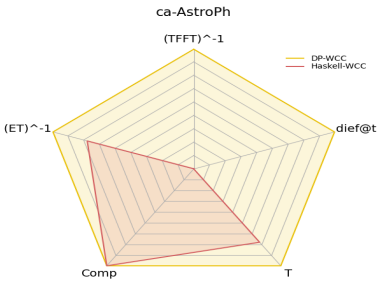


Figure 8: ca-AstroPh Dm

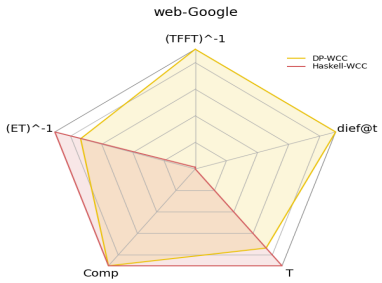


Figure 9: web-Google Dm

As we can appreciate in the above radar plots our previous analysis can be confirmed. We can see for example that the Throughput of *web-Google* in Figure 9, in the case of DP-Haskell is worse than Haskell `containers`, which is not happening for the others.

In conclusion, we can say that regarding [Q2] (section 5) although DP-Haskell is faster than the traditional approach, the speed-up dimension execution factor is not always the most interest analysis that we can have, because as we have seen even when in the case of *web-Google* Graph DP-Haskell

is slower at execution, it is at least generating incremental results without the need to wait for the rest of the computations.

6.3. Experiment: E3

For this type of analysis, our experiment focuses on *email-Enron* network [28] only because profiling data generated by GHC is big enough to conduct the analysis and on the other, and enabling profiling penalize execution time.

Multithreading. For analyzing parallelization and multithreading we have used *ThreadScope* [36] which allows us to see how the parallelization is taking place on GHC at a fine grained level and how the threads are distributed throughout the different cores requested with the `-N` execution `ghc-option` flag.

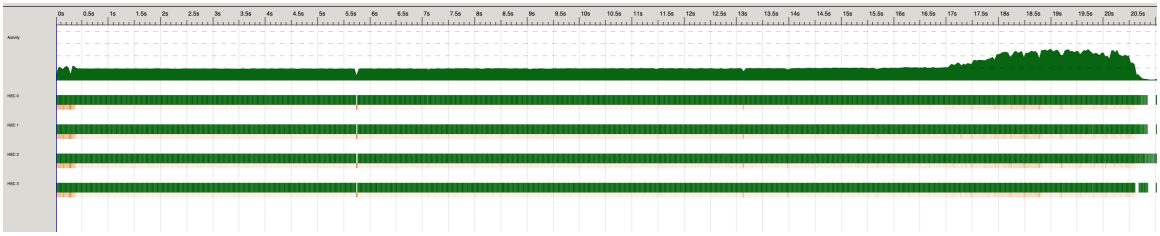


Figure 10: Threadscope Image of General Execution

In Figure 10, we can see that the parallelization is being distributed evenly among the 4 Cores that we have set for this execution. The distribution of the load is more intensive at the end of the execution, where `actor2` filter stage of the algorithm is taking place and different filters are reaching execution of that second actor.

Another important aspect shown in Figure 10, is that this work is not so significant for GHC and the threads and distribution of the work keeps between 1 or 2 cores during the execution time of the `actor1`. However, the usages increase on the second actor as pointed out before. In this regard, we can answer research questions [Q1] and [Q3] (section 5), verifying that Haskell not only supports the required parallelization level but is evenly distributed across the program execution too.

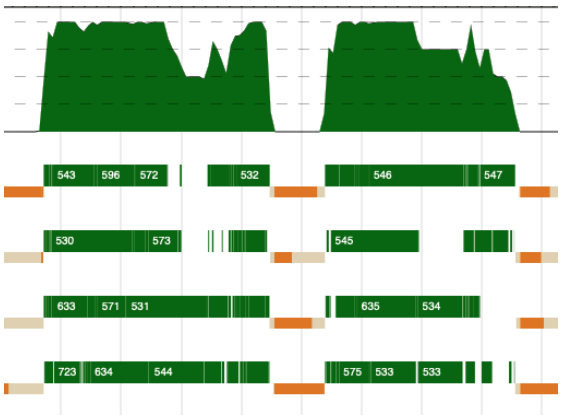


Figure 11: Threadscope Image of Zoomed Fraction

Finally, it can also be appreciated that there is no sequential execution on any part of the program because the 4 cores have *CPU* activity during the whole execution time. This is because as long the program start, and because of the nature of the DPP model, it is spawning the *Source* stage in a separated thread. This is a clear advantage for the model and the processing of the data since the program does not need to wait to do some sequential processing like reading a file, before start computing the rest of the stages.

Figure 11 zooms in on *ThreadScope* output in a particular moment, approximately in the middle of the execution. We can appreciate how many threads are being spawned and by the tool and if they are evenly distributed among cores. The numbers inside green bars represent the number of threads that are being executed on that particular core (horizontal line) at that execution slot. Thus, the number of threads varies among slot execution times because as it is already known, GHC implements *Preemptive Scheduling* [6].

Having said that, it can be appreciated in Figure 11 our first assumption that the load is evenly distributed because the mean number of executing threads per core is 571.

Memory allocation. Another important aspect in our case is how the memory is being managed to avoid memory leaks or other non-desired behavior that increases memory allocation during the execution time. This is even more important in the particular implementation of WCC using DPP model because it requires to maintain the set of connected components in memory throughout the execution of the program or at least until we can output the calculated WCC if we reach to the last *Filter* and we know that this WCC cannot be enlarged anymore.

In order to verify this, we measure memory allocation with *eventlog2html* [22] which converts generated profiling memory eventlog files into graphical HTML representation.

As we can see in Figure 12, DP-Haskell does an efficient work on allocating memory since we are not using more than 57 MB of memory during the whole execution of the program.

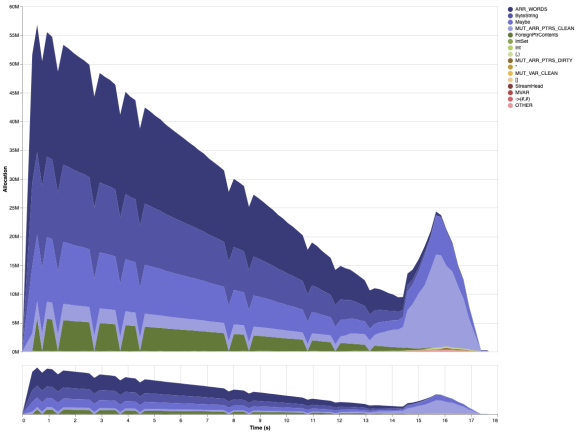


Figure 12: Memory Allocation

On the other hand, if we analyze how the memory is allocated during the execution of the program, it can also be appreciated that most of the memory is allocated at the beginning of the program and steadily decrease over time with a small peak at the end that does not overpass even half of the initial peak of 57 MB. The explanation for this behavior is quite straightforward because at the beginning we are reading from the file and transforming a `ByteString` buffer to `(Int, Int)` edges. This is seen in the image in which the dark blue that is on top of the area is `ByteString` allocation. Light blue is allocation of `Maybe a` type which is the type that is returned by the *Channels* because it can contain a value or not. Data value `Nothing` is indicating end of the *Channel*.

Another important aspect is the green area which represents `IntSet` allocation, which in the case of our program is the data structure that we use to gather the set of vertices that represents a WCC. This means that the amount of memory used for gathering the WCC itself is minimum and it is decreasing over time, which is another empirical indication that we are incrementally releasing results to the user. It can be seen as well that as long the green area reduces the lighter blue (`MUT_ARR_PTRS_CLEAN` [25]) increases at the same time indicating that the computations for the output (releasing results) is taking place.

Finally, according to what we have stated above, we can answer the question [Q3] (section 5) showing

that not only memory management was efficient, but at the same time, the memory was not leaking or increasing across the running execution program.

7. Related Work

Several implementations for streaming processing models [18, 30, 35] in Haskell have arisen over the years. All these libraries have their abstractions and can do data streaming processing in a fast way with different performance according to recent benchmarks [16]. Although they seem to be suitable for implementing a DP, it is required to know pipeline stages disposition beforehand, and it is hard to achieve a succinct and expressive implementation of a DPF. Moreover, since they have been conceived as a data parallel streaming model [12] by design instead of pipeline parallel streaming, implementing DP using these tools becomes counter-intuitive and hard to achieve.

Another kind of streaming implementation in Haskell is described in [7]. In that work, the author describes how to encode pipeline parallelism with `Par` *Monad*. Although this could have been a suitable alternative for implementing DP, the parallelization level used by `Par` *Monad* is sparks [32]. As we have explained in section section 3, we do not require to reach that level of parallelization in our current model.

In regards to other DP language implementations, a significant contribution on [10] has been done, where a DP implementation in Go Programming Language (Go) for counting triangles of graphs is compared against MapReduce. Those experiment results have shown how DP in Go improves the performance in terms of execution time and memory depending on the graph topology. It would be interesting and a matter of future work, to compare different language implementations of DP*s*, taking into consideration those promising results and the ones presented in this article.

8. Conclusions and Ongoing Work

The empirical evaluation of the DP-Haskell implementation to compute weakly connected components of a graph, evidence suitability, and robustness to provide a Dynamic Pipeline Framework in that language. Measuring using dief@t metrics in section 6.2 reveals some advantageous capability of $\mathsf{DP}_{\mathsf{WCC}}$ implementation to deliver incremental results compared with default containers library implementation. Regarding the main aspects where DPP is strong, i.e. pipeline parallelism and time processing, the $\mathsf{DP}_{\mathsf{WCC}}$ performance shows that Haskell can deal with the requirements for the WCC problem without penalizing neither execution time nor memory allocation. In particular, the $\mathsf{DP}_{\mathsf{WCC}}$ implementation outperforms in those cases where the topology of the graph is more sparse and where the number of vertices in the largest WCC is not big enough. We think this work has gathered enough evidence to show that the implementation of Dynamic Pipeline in Haskell Programming Language is feasible. This fact opens a wide range of algorithms to be explored using the Dynamic Pipeline Paradigm, supported by purely functional programming language. As we mentioned in section section 3, we are addressing the design and the definition of the DSL in Haskell, taking into account the knowledge obtained in this work. The complete DPF's implementation will contain the *Type-Level* DSL allowing the user to define algorithms in terms of DP and the Interpreter of DSL (IDL) that will be mainly based on what has been presented here.

References

[1] M. Acosta, M.-E. Vidal, and Y. Sure-Vetter. Diefficiency metrics: Measuring the continuous efficiency of query processing approaches. pages 3–19, 10 2017. ISBN 978-3-319-68203-7. doi: 10.1007/978-3-319-68204-4_1.

[2] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 62–73, 2013. doi: 10.1109/ICDE.2013.6544814. URL `http://dx.doi.org/10.1109/ICDE.2013.6544814`.

[3] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGOPS Operating Systems Review*, 40(5):151–162, 2006.

[4] T. Harris, S. Marlow, and S. Peyton Jones. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM Press, January 2005. ISBN 1-59593-080-9. URL `https://www.microsoft.com/en-us/research/publication/composable-memory-transactions/`.

[5] I. Lee, T. Angelina, C. E. Leiserson, T. B. Schardl, Z. Zhang, and J. Sukha. On-the-fly pipeline parallelism. *ACM Transactions on Parallel Computing*, 2(3):17, 2015.

[6] P. Li, S. Marlow, S. Peyton Jones, and A. Tolmach. Lightweight concurrency primitives for ghc. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, page 107–118. Association for Computing Machinery, 2007. ISBN 9781595936745. doi: 10.1145/1291201.1291217.

[7] S. Marlow. Parallel and concurrent programming in Haskell. In V. Zsók, Z. Horváth, and R. Plasmeijer, editors, *CEFP 2011*, volume 7241 of *LNCS*, pages 339–401. O'Reilly Media, Inc., 2012.

[8] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. *SIGPLAN Not.*, 46(12):71–82, 2011. ISSN 0362-1340. doi: 10.1145/2096148.2034685.

[9] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 124–144, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. ISBN 978-3-540-47599-6.

[10] E. Pasarella, M. Vidal, and C. Zoltan. Comparing mapreduce and pipeline implementations for counting triangles. In A. Villanueva, editor, *Proceedings XVI Jornadas sobre Programación y Lenguajes, PROLE 2016, Salamanca, Spain, 14-16th September 2016*, volume 237 of *EPTCS*, pages 20–33, 2016. doi: 10.4204/EPTCS.237.2. URL `https://doi.org/10.4204/EPTCS.237.2`.

[11] T. Petricek. What we talk about when we talk about monads. *The Art, Science, and Engineering of Programming*, 2(3), 2018. ISSN 2473-7321. doi: 10.22152/programming-journal.org/2018/2/12.

[12] H. Röger and R. Mayer. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Computing Surveys (CSUR)*, 52:1 – 37, 2019.

[13] P. Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, page 24–52, Berlin, Heidelberg, 1995. Springer-Verlag. ISBN 3540594515.

[14] C. Zoltán and et al. The dynamic pipeline paradigm. In *Jornadas sobre Programación y Lenguajes. Actas de las XIX Jornadas de Programación y Lenguajes (PROLE 2019): Cáceres,*

septiembre de 2019, pages 1–11, San Sebastián, Guipúzcoa: Sociedad de Ingeniería de Software y Tecnologías de Desarrollo de Software (SISTEDES), 2019.

[15] Simon Marlow. Haskell async library. `https://hackage.haskell.org/package/async`. Accessed: 2021-04-17.

[16] Harendra Kumar. Haskell stream libraries benchmarks. `https://github.com/composewell/streaming-benchmarks`. Accessed: 2021-05-09.

[17] D. Stewart, D. Coutts. Haskell bytestring library. `https://hackage.haskell.org/package/bytestring`. Accessed: 2021-04-17.

[18] Michael Snoyman. Haskell conduit types. `https://hackage.haskell.org/package/conduit`. Accessed: 2021-05-09.

[19] Haskell.org. Haskell containers library. `https://hackage.haskell.org/package/containers`. Accessed: 2021-04-17.

[20] Bryan O'Sullivan. Haskell criterion tool. `https://hackage.haskell.org/package/criterion`. Accessed: 2021-04-17.

[21] Maribel Acosta. Python diefficiency metric tool. `https://github.com/SDM-TIB/diefpy/`. Accessed: 2021-05-03.

[22] M. Pickering, D. Binder, C. Heiland-Allen. Haskell eventlog2html tool. `https://mpickering.github.io/eventlog2html/`. Accessed: 2021-04-17.

[23] Glasgow Haskell Compiler. Glasgow haskell compiler user's guide. `https://downloads.haskell.org/ghc/8.8.4/docs/html/users_guide/glasgow_exts.html#language-options`. Accessed: 2021-05-09.

[24] Haskell.org. Haskell base library. `https://hackage.haskell.org/package/base-4.15.0.0/docs/Control-Concurrent.html#v:forkIO`. Accessed: 2021-04-26.

[25] Haskell.org. Haskell ghc-heap types. `https://downloads.haskell.org/~ghc/8.10.4/docs/html/libraries/ghc-heap-8.10.4/GHC-Exts-Heap-ClosureTypes.html`. Accessed: 2021-05-09.

[26] S. Marlow, R. Newton. Haskell monad-par library. `https://hackage.haskell.org/package/monad-par`. Accessed: 2021-05-11.

[27] J. Leskovec, J. Kleinberg and C. Faloutsos. Snap datasets: Stanford large network dataset collection. https://snap.stanford.edu/data/ca-AstroPh.html, 2007.

[28] William Cohen. Snap datasets: Stanford large network dataset collection. https://snap.stanford.edu/data/email-Enron.html, 2004.

[29] Google Inc. Snap datasets: Stanford large network dataset collection. https://snap.stanford.edu/data/web-Google.html, 2002.

[30] Gabriel Gonzalez. Haskell pipes library. `https://hackage.haskell.org/package/pipes`. Accessed: 2021-05-09.

[31] D. Kovanikov, V. Romashkina, S. Diehl, Serokell. Haskell relude library. https://hackage.haskell.org/package/containers. Accessed: 2021-04-17.

[32] Glasgow Haskell Compiler. Glasgow haskell compiler user's guide. https://downloads.haskell.org/ghc/8.8.4/docs/html/users_guide/parallel.html#annotating-pure-code-for-parallelism. Accessed: 2021-04-28.

[33] FP Complete. Haskell stack tool. https://docs.haskellstack.org/en/stable/README/. Accessed: 2021-05-11.

[34] Stanford University. Snap datasets: Stanford large network dataset collection. https://snap.stanford.edu/data/index.html. Accessed: 2021-04-15.

[35] Harendra Kumar. Haskell streamly types. https://hackage.haskell.org/package/streamly. Accessed: 2021-05-09.

[36] S. Singh, S. Marlow, D. Jones, D. Coutts, M. Konarski, N. Wu, E. Kow. Haskell threadscope tool. https://wiki.haskell.org/ThreadScope. Accessed: 2021-04-17.

[37] Brandon Simmons. Haskell unagi-chan library. https://hackage.haskell.org/package/unagi-chan. Accessed: 2021-04-17.