Modern Parallelization for a Dataflow Programming Environment

Zackary Kurima-Blough[1], **Mark C. Lewis**[1], **and Lisa Lacher**[2]
[1]Department of Computer Science, Trinity University, San Antonio, Texas, USA
[2]College of Science and Engineering, University of Houston at Clear Lake, Clear Lake, Texas, USA

Abstract— *We explore a number of options to process a dataflow programming environment in parallel to potentially use as part of an upgrade for SwiftVis, a dataflow programming tool for data analysis and plotting used primarily in planetary science. The implementation that we found to be most appropriate, using composable futures, was implemented and we present performance results showing that it scales very well with thread count for a sample graph. The approach of using composable futures also results in code that is very clean and doesn't contain any custom synchronization or blocking which are aspects of code that often lead to bugs in multithreaded code.*

Keywords: Multithreading, Futures, Dataflow programming, Actors, Reactive Streams

1. Introduction

SwiftVis[13] is a data analysis and visualization tool originally written in 2001-2004 with support from a NASA AISRP grant to assist planetary scientists working with data produced by the Swift N-Body simulation code. It was created as part of a more general upgrade of the SWIFT [11] simulation suite to a version called Swifter [12]. There were a few specific goals for the design of SwiftVis that had a significant impact on its construction, and which would have to be met in any redesign. Here is a list of the main requirements along with some motivation/description.

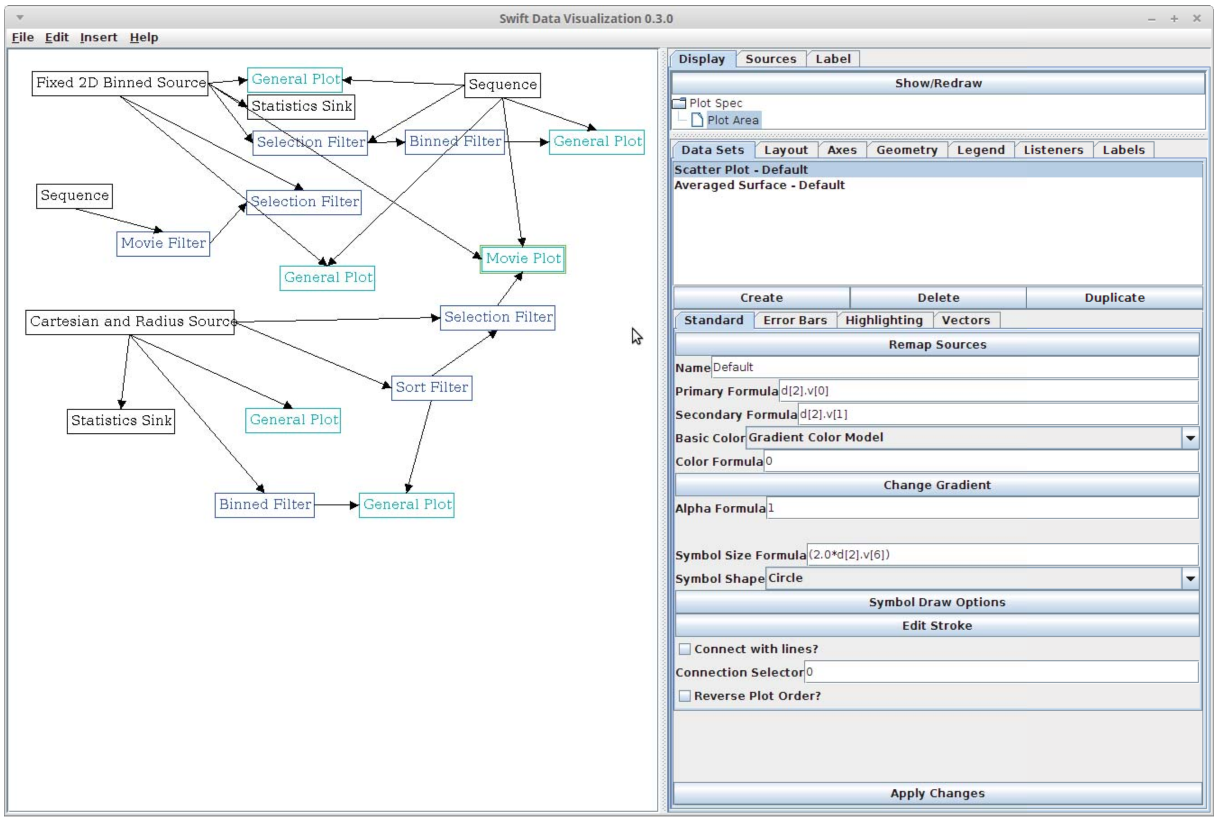
- No traditional, text-based programming required: While most planetary scientists have some programming ability, it is often in somewhat niche languages, like Fortran or IDL, and their comfort level with code is often limited.
- Large data: While the data sets of the early 2000s were small by today's standards, many were created by long simulations run on clusters of machines and their handling needed to be done efficiently for the tool to be usable.
- Allow "playing" with data: This involves giving users the ability to change things and immediately see results in the output. It also extends to having the ability to make movies/animations of systems over time or as some other parameter is changed.
- Publication quality output: While it would be possible to use this tool to make discoveries and then move to another tool to make final versions of plots, the usefulness of the tool is greatly enhanced if it can produce output of sufficient quality for publication itself.

The design that was selected to meet these requirements for SwiftVis was to use a dataflow graph where data is read in to "sources" then passes through various "filters" before ending up at "sinks". A screen shot of a sample usage of SwiftVis is shown in figure 1.The design made it fairly easy to create additional types of sources, filters, and sinks. The system also supports dynamic loading of compiled Java classes so that users can create their own subtypes and use them as well. Over the years many types of sources and filters were created, but there are only two types of sinks, a plot and an element that displays key statistics on the input.

The various types of sources that have been created pull data in from a number of different planetary/astrophysical simulation packages (Mercury, Symba, HNBody, PDKGrav) as well as other common text format files. While SwiftVis started life with a variety of low-level filters that could be combined in various ways to do a variety of tasks, many more filters of various complexities have been added over time.

This dataflow programming design produced a tool that was far more generally applicable than the original goal of just working with Swift output data. It has also been used extensively with N-body rings simulations, which have a fundamentally different nature in that they run for a lot fewer orbits, but typically include many orders of magnitude more particles. The scientists who use SwiftVis often use it as their general plotting tool when working with text files and other data as well. In general, any numeric data can be manipulated with the tool as long as there is a source to load in.

In addition to adding new types of sources and filters, SwiftVis was also extended in two other significant ways after the original implementation was written. One extension was to let various elements buffer data so that really large data sets did not have to reside in memory all the time. A second extension was to add parallelism, both at the graph level, and in individual elements. As often happens with features added after the initial design and creation of a software package, these features introduced some rare, but significant, bugs into the code. This paper documents our work exploring alternative approaches to the graph parallel aspects of SwiftVis to see if we could use more modern parallel constructs in order to simplify this code, and reduce the prospect of errors.



Scatter Plot - Default
Averaged Surface - Default

Name Default
Primary Formula d[2].v[0]
Secondary Formula d[2].v[1]
Basic Color Gradient Color Model
Color Formula 0
Alpha Formula 1
Symbol Size Formula (2.0*d[2].v[6])
Symbol Shape Circle
☐ Connect with lines?
Connection Selector 0
☐ Reverse Plot Order?

Fig. 1
THIS SHOWS THE SWIFTVIS INTERFACE AS IT CURRENTLY EXISTS. THE DATAFLOW DAG IS SHOWN ON THE LEFT SIDE WHERE THE NODE LABELED "MOVIE PLOT" IS HIGHLIGHTED. ON THE LEFT ARE THE SETTING FOR THAT NODE.

1.1 Dataflow Programming

In dataflow programming, the data "flows" through a program, being modified and stored along the way. It is a programming paradigm that models a program as a directed graph [2], [5] in which the nodes represent process instructions, such as comparison operations or arithmetic, and the arcs represent the data flows and dependencies between the processes [10]. The arcs that flow toward a node are considered the input, while the ones that flow away from the node are considered the output. The data flows along the arcs which behave like unbounded FIFO queues [9]. When the program begins, data is placed onto key input arcs by activation nodes and this triggers the rest of the program. When a node's input arcs have data on them, then that node is said to be *fireable* [3] and is later executed at some undefined time. When executed, it removes the data from the input arcs, performs its operation, and places new data on some or all of its output arcs and then ceases execution and waits to become fireable again. The key advantage of the dataflow paradigm is that more than one instruction can be executed at once. Because of this, if several instructions become fireable at the same time, then they can be executed in parallel. Thus, it can be seen that the dataflow method provides the potential for tremendous parallel execution at the instruction level.

The original motivation for research into dataflow was the desire to exploit the opportunities that massive parallelism might provide. It was thought that the conventional "von Neumann" processors were unsuited for parallelism because bottlenecks caused by its mutable global memory and its global program counter [1]. An alternative to this was the dataflow architecture, which initially looked very promising [2], [5]. However, researchers found problems trying to use conventional imperative programming languages on the dataflow hardware; particularly programs associated with locality and side-effects [10], [2]. This resulted in the creation of so-called dataflow programming languages that restricted certain aspects, such as assignments, and compiled into dataflow graphs, which are considered the "machine language" of dataflow computers [1], [18], [3]. While dataflow programs can be expressed graphically, traditional dataflow languages are primarily text-based. This is primarily because the hardware for displaying graphics had not been available at the time. Although dataflow languages were initially designed for hardware that incorporated the dataflow architecture, the use of dataflow programming languages is not limited to dataflow machines. The best list of features that constitute a dataflow languages was put forth by Ackerman [1]. This list includes the following:

- freedom from side effects
- locality of effect
- data dependencies equivalent to scheduling
- single assignment variables
- lack of history sensitivity in procedures

It is clear from the above that dataflow languages are basically functional; however, this does not mean that dataflow and functional languages are equivalent. It is possible to write programs in a functional language which cannot be implemented as a dataflow graph [8]. Also, much of the syntax of dataflow languages (such as loops) comes from imperative languages. Thus, dataflow languages are basically functional languages with an imperative syntax[8].

In the 1970s and early 1980s, many thought that dataflow architecture would surpass the von Neumann architecture [16], [17]. However, this never really happened because the parallelism that was used in dataflow architectures proved to be too fine-grained to be effectively distributed and better performance was found through hybrid von Neumann dataflow architectures. These hybrid architectures comprise most of the current dataflow architecture efforts and many of these take advantage of more coarse-grained parallelism which groups together a number of dataflow instructions and then executes them in sequence [8].

In the 1990s, the field of dataflow visual programming languages began to grow [4], [13], [18], [6], [7]. Some of these have become successful commercial products, such as LabView, whereas others, such as NL and SwiftVis are still primarily used for research. Although dataflow visual programming languages have traditionally been concerned with the ability to exploit parallelism, and this remains an important requirement, many of today's dataflow visual programming languages key advantages lie with the software development lifecycle [8].

2. Approach

The original SwiftVis implementation was written in Java. This had one huge advantage at the time in that it enabled a graphical application to be run across multiple platforms with a single code base and didn't require users to recompile on their own machines. That particular benefit mainly comes from the JVM, not the Java language itself. With the rewrite, we are considering moving over to Scala for two primary reasons. The first is that we want to include a scripting environment, which Scala comes with, but Java does not. The desire to have a scripting environment stems from the fact that certain types of porcessing, like batch processing, simply work better in a scripting environment than in a graphical one. The second reason for switching to Scala, which is more relevant here, is that Scala has richer, more robust support for parallelism than Java currently does. While the parallelism support in Java libraries can improve, the language itself can't be modified to match the expressivity of Scala, which has a significant impact on how easy it is to include parallelism in the code.

The graph level parallelism in the original SwiftVis was written using constructs from `java.util.concurrent` such as `ExecutorService`s with code that analyzed the graph so that when a change was made, all nodes that would be impacted by the change were identified and added to a list for processing, then all nodes that needed to be processed, but which didn't have any inputs that still needed to be processed were started. Any time a node finished processing, it would notify the scheduler, which would repeat the process of looking for safe nodes to process and start those. Every part of this process, including modification of the graph, involved significant mutations, which could lead to race conditions.

We explored three main alternatives to this basic threading approach that use different higher-level parallelism libraries: composable futures, actors, and reactive streams. This paper focuses on the composable futures approach, but we will briefly describe the motivation behind the other options and why we eventually decided against using them.

One of the key considerations when looking at potential implementations is that the graph might be used in either a scripting environment or a graphical environment, as the addition of a scripting environment is one of the major motivations behind considering a rewrite for SwiftVis. These two environments have some key differences in their use cases. In the graphical environment, it is very likely that users will try to modify the graph while parts of it are processing. When this happens, partial results in one part of the graph can lead to errors when a different part of the graph propagates its changes. This type of usage is extremely unlikely in a scripting environment, but the scripting environment has a different challenge in that users might want to actively use different "versions" of a graph. That is to say that they might build a graph that does something they like, then use that graph as the foundation for a modified graph that they process, and the processing of both might happen at the same time, potentially leading to a different type of race condition. This situation is effectively impossible in the GUI environment as the user only has access to the current graph that is being displayed. An effective solution needs to deal well with both of these situations.

2.1 Actors

This project began with the idea that each element in the dataflow graph could be represented as an actor using the actor model of parallelism, specifically the Akka library. The motivation behind this is that actors not only handle parallelism, but they also make it much easier to have safe mutable state as each actor only uses a single thread, and that thread can freely mutate any data that is known only to that actor. As mutable state is the primary cause of errors in the existing implementation and in the challenging situations described above, the ability to handle it in a nearly transparent way was highly motivating.

Unfortunately, actors don't naturally provide a good mechanism for scheduling graph elements to run in a manner such that no element runs before all of its inputs have been processed. We considered several options for passing messages downstream, then back upstream through the graph to produce the desired results, but all such approaches included elements that might not behave as intended in certain extreme conditions where the timing of message sending and processing was different than expected. This was particularly true is situations where the graph was part of a GUI and users could modify parts of it while other parts were processing.

In addition, while the actors help confine mutable state in a way that is useful for the GUI-based use case, they don't automatically provide any mechanisms for dealing with the issues of utilizing progenitor graphs in any way that might be helpful for the scripting environment.

Indeed, in the discussions where we tried to design a good actor-based approach, we finally came to the conclusion that the ideal implementation would actually involve an immutable graph, where modifications would produce new graphs that reuse components of the previous graph. This more functional type of structure helps to fix many of the challenges with both the GUI and scripting based interfaces, but introduces a number of its own challenges that are discussed in 2.4.

2.2 Reactive Streams

Another approach that seems to be a natural fit is that of reactive streams. The reactive stream standard[14] is a modern specification, published in 2015, for handling streaming data. There are a number of different implementations of the reactive streams standard. We looked primarily at the implementation in Akka, called Akka Streams. Part of the reason that streams seem like a natural fit is that they are built to do parallel processing on data using a graph based formalism. A key component to reactive streams is back pressure. This is a feature where components in the data flow talk to one another to control the rate at which data is sent to different components to allow the queues to be bounded.

Unfortunately, there isn't perfect fit between Akka Streams and our application either. One of the key differences is that we aren't dealing with streaming data, and as such, back pressure doesn't provide a significant benefit. So, as with the actors where we decided that we don't really need the encapsulated mutability, the primary goal that we are looking for is just manageable parallelism.

There were three main challenges with using Akka Streams for our purposes. One is that the nodes in the graph for Akka Streams can have one input and one output, multiple inputs and one output, or one input and multiple outputs. It doesn't have a natural way to support a node with multiple inputs and multiple outputs. That is problematic, but we could get around it my modeling out multiple input, multiple output nodes as two nodes stuck together. The other two issues are specifically problems for the graphical interface. One issue is caused by the fact that you can't arbitrarily change the structure of the graph after creation. The other issue is the fact that the graph has to be "closed" in order to run. This requirement means that all inputs and outputs from the graph have to be fully specified before it can be run, a requirement that doesn't match well with the requirements of an interactive graphical application.

2.3 Composable Futures

The implementation that we decided upon, and which we present results for in section 3, is to use composable futures. Futures have become standard features of parallel libraries in most languages and represent potentially uncompleted computations. Historically, the primary ways of dealing with futures were to either block and wait for their results, or to schedule callbacks that will be executed when the future is completed. The blocking option is undesirable as it generally causes code to be less parallel. Well designed multithreaded code avoids blocking to maximize thread utilization. Using callbacks leads to poorly structured code[1].

The solution to blocking and callbacks is to make it so that futures include an applicative function that executes code when the future has completed, much like a callback, but which gives back a new future immediately so that calculations can be chained and combined instead of nesting them in callbacks. In Scala, the language used for our implementation, the `scala.concurrent.Future` type uses `map` as the applicative function.

Another key value of applicative functions is that they can be combined in various ways. For the purposes of this work, it is extremely useful to be able to combine a sequence of future objects into a single future that produces a sequence when all the individual futures have completed. The `Future` companion object contains a number of methods for combining `Future`s and the `Future.sequence` method specifically converts a `Seq[Future[A]]` to a `Future[Seq[A]]`. This method is used to combine the inputs to a filter, represented as a `Seq[Future[A]]`, into a single future object that can then be mapped to the function of the node.

[1]The frequent use of callbacks for asynchronous processes in JavaScript is so notorious that it is often referred to as "callback hell".

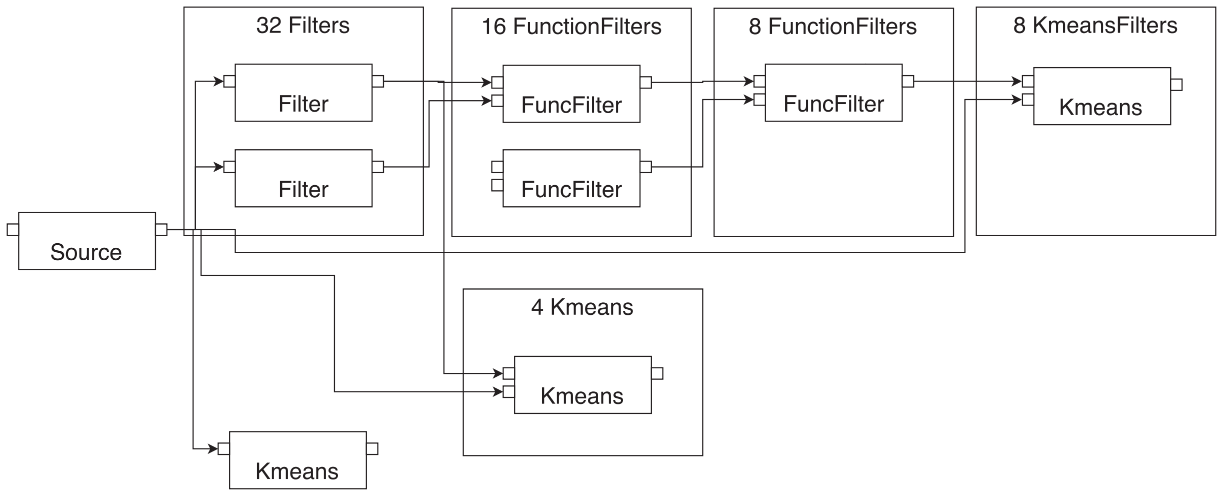


Fig. 2
THIS IS A GRAPHICAL REPRESENTATION OF THE DATAFLOW GRAPH USED FOR PERFORMANCE TESTING IN THIS WORK. IT HAS MULTIPLE LAYERS WITH A LARGE BRANCHING FACTOR TO MAKE IT FEASIBLE FOR MULTIPLE THREADS TO IMPROVE PERFORMANCE. THE K-MEANS FILTERS WERE SELECTED AS THEY ARE NUMERICALLY INTENSIVE, BUT GENERALLY APPLICABLE TO MANY DATA SETS.

That procedure effectively describes the approach used here. The graph is traversed, and as every node is visited, a future is created that maps the function of that node on the results of the inputs. All the results are wrapped in futures, and the function for that node is mapped to the result. The traversal is done as a memoized recursive function, though it could also be done as a simple traversal of a topological sort of the graph. This implementation winds up being remarkably simple, but generally meets the needs of both the scripting environment and the GUI environment.

2.4 Immutable Graphs

One key aspect of the design that we eventually settled on is the decision that all graphs would be immutable, and that operations that modify the graph do so by creating new graphs. To make this more efficient, so that elements could be shared from the original graph to the product graph, the nodes are named, and accessed through a `Map`. In addition, the edges are stored in `Map`s that only reference the keys. The Scala libraries include immutable `Map` types that are well suited for this purpose and efficiently use memory by sharing parts of the structure when modifications are made, so the revised versions of connections when nodes or edges are added or removed don't consume too much memory. Not having edges stored as references in the nodes also allows all the nodes to be shared. The data that is the output of the nodes is stored in a separated state, and for the composable future implementation, they are wrapped in futures. When a change is made, all the futures for the resulting data are preserved, and shared with the old graph, except for those that are downstream from the change. These are cleared out of a `Map` so that when the graph is run, they will be recalculated, while any values that should be unchanged can simply be referenced.

There are a few other details of the implementation that are worth mentioning. One is that we keep two kinds of graphs. The standard graph type has methods that modify the graph by adding nodes or edges. It also has a method which we called `run` that executes the graph. This method returns a `RunGraph` that extends the basic graph, but also includes a method to access the data. Normal graphs can potentially store data, but the data is not complete, and would not reflect any changes since the last time a graph had been run. The `RunGraph` type represents a state after run has been called when all the data is potentially available. We use the word potentially here because in the implementation with futures, the calculation might still be in progress, but it is still retrievable either with a blocking call or wrapped in a future.

While it wasn't designed for it, this implementation also does a very good job of not duplicating work when a graph that is currently processing is modified and run. Modification of a `RunGraph` returns a normal graph that has a `Map` of the futures for all the results that would not be changed by that modification. If the modified graph is run, the resulting `RunGraph` will use all of the already existing futures of results, which might still be processing, and only create new futures for results that depend on the alteration. This type of behavior is particularly beneficial in a GUI, where users will often choose to have the graph reevaluate automatically with every change.

The primary potential drawback of the immutable graphs is their use in the scripting environment. Anyone using the scripting environment who does not have a background in functional programming will have to adjust their programming style a bit to deal with the fact that most operations make new graphs instead of mutating a single graph. We have considered the possibility of adding a `GraphWrapper` or something similar that acts like a mutable graph for scripting purposes, but we have not yet explored this option extensively.

3. Performance Results

We implemented a sequential version of our data flow graphs, as well as one that used composable futures using the approach described above. The sequential version worked by doing a topological sort on the elements in the graph and processing them in the order of that sort. This fairly simple approach provides a speed baseline to measure the overhead introduced by parallelism, as well as for general comparison to make sure that we weren't doing anything wrong with the implementations[2].

For testing purposes, we coded up the graph shown in figure 2[3]. This graph begins by running the data through 32 different filters that divide the data between equally spaced bins using one value from the input with a broad range of values. Pairs of outputs from those were then run through functions that made minor modifications to the data elements in two different stages. These modifications were not meaningful, we simply replaced one of the values with its square, both to provide extra processing work and to alter the input for clustering purposes. The use of two inputs was significant in that it helped show that the ordering of the graph processing was valid. At the end of all paths, data that includes the full original set is run through a k-means algorithm. The k-means was selected because it is time consuming, which is good for timing, but also very generally applicable to a wide variety of types of data.

It is also worth noting that for this work, none of the individual filters were parallelized, as we want to focus on the impact of the dataflow parallelism, not whatever parallelism we can put into the filtering, function applications, or k-means calculations. A full implementation would certainly seek to exploit parallelism at all levels, but such an implementation would hide what we are interested in exploring here.

[2]This proved useful as there were subtle errors in the original construction of the testing graph that we wouldn't have noticed were it not for the timing discrepancies between the sequential and multithreaded versions.

[3]Note that this was written in code, not drawn in an interface like that show in in figure 1 for SwiftVis.

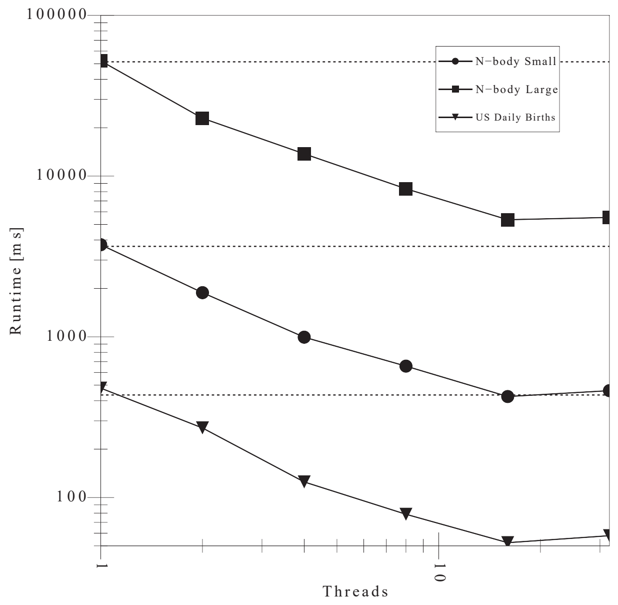


Fig. 3

THIS SHOWS THE TIMING RESULTS FOR THREE DIFFERENT DATA SETS USING THE GRAPH SHOWN IN FIGURE 2. THE N-BODY DATA SETS WERE FROM NUMERICAL SIMULATIONS OF PLANETARY RINGS AND HAD 18,000 AND 180,000 DATA POINTS IN THE LARGE AND SMALL SETS RESPECTIVELY. THE BABY NAMES DATA SET INCLUDED A FEW THOUSAND DATA VALUES. THE DOTTED LINES SHOW THE TIMING OF THE SEQUENTIAL GRAPH IMPLEMENTATIONS, WHICH WERE VERY CLOSE TO THE MULTITHREADED IMPLEMENTATIONS USING ONLY A SINGLE THREAD.

Figure 3 shows data points for the timing results from the composable future based implementation as well as dotted lines for the sequential version[4]. The timing results come from three different data sets with two different types of data. The smallest data sets only used a few thousand data elements, that came from a data set looking at the number of babies born in the United States over time. Each data element contained a value for year, month, day of month, day of week, and the number of babies born on that day. With the exception of the counts, the other values are integers with fairly small ranges, so there are lots of overlaps, which impacts the rate of convergence of the k-means clustering algorithm. This is part of the reason this data set processed so quickly on our test graph.

The other two data sets are particle data from a large simulation of planetary rings. Only a certain fraction of the particles in the simulation were used. For the smaller, just over 18,000 particles were used, and for the larger one, just over 180,000 particles were used. Each particle had positions

[4]Note that this plot was made using the original SwiftVis.

and velocities in 3D, so there were six floating point values with basically no ties. This made convergence for k-means slower.

The tests were run on a machine with two AMD Opteron 6272 processors and 32 GB of RAM, configured with 16 GB next to each processor socket. Each Opteron 6272 has 16 cores and no symmetric multithreading. Timing results were calculated using the ScalaMeter performance benchmarking package[15], and the times were stable at the level of a few percent. On this hardware, the sequential version ran at roughly the same speed as the threaded version using a single thread. Also, all three data sets showed roughly linear scaling from 1 to 4 threads with slightly less than linear scaling as the thread count went up to 8 and 16. Looking back at figure 2, you can see that there are only 13 k-means filters, and that the longest paths go down to being eight filters across, so the graph itself limits the parallelism somewhat above 8 threads. Across all data sets, the 16 thread test ran roughly 9x faster than the single thread test or the sequential test.

While going to 32 threads slowed the processing on all data sets, this is likely due to the nature of the hardware. While the machine has 32 cores, it is configured in two processors with 16 each. The 16 cores in each processor have shared cache and fast access to a single 16 GB memory bank. When more than 16 threads are being used, some of the work has to be moved to a separate processor with its own cache and local memory. Given that in this workload the amount of work that can be done in more than 13 threads is small, it is not surprising that spinning up an additional 16 threads to work on it causes enough extra work in the memory system to slow the full processing down.

4. Conclusions

The general conclusion of this work is that composable futures work very well for for processing tasks that can be modeled as acyclic graphs such as the dataflow programming considered here. Our test results show that this approach has very low overhead and scales to high thread counts as long as there is sufficient parallelism present in the dataflow graph.

The primary motivation for exploring this was not performance improvements, but to find a model that was less bug prone. We feel that the composable futures approach, combined with immutable graph representations does a good job here as well. The implementation for this has no explicit synchornization or other forms of blocking. When a graph is run, we simply run through the graph nodes recursively, making futures that calculate the results of that note, mapping them on the futures of any inputs to that filter. This way, the code in the futures automatically handles proper sequencing. We don't even perform a topological sort, though one could do so instead of using memoized recursion.

While we are satisfied for the approach of using an immutable graph and scheduling work using composable futures to create parallelism, there are still other features of SwiftVis that we want to explore improving upon as part of our rewrite. These include a better user interface with more robust and usable error/warning reporting. We would like graphs built either through scripting or in the GUI to be savable in a manner that allows them to be loaded, manipulated, and run using the other interface. We also want to explore speeding up the rendering of plots using JavaFX instead of Swing. Lastly, we want to explore the possibility of integrating SwiftVis with distributed, big data frameworks, like Spark.

References

[1] W. B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, Feb. 1982.
[2] Arvind, , and D. E. Culler. Dataflow architectures. *Annual Review of Computer Science*, 1(1):225–253, 1986.
[3] Arvind and D. E. Culler. Annual review of computer science vol. 1, 1986. chapter Dataflow Architectures, pages 225–253. Annual Reviews Inc., Palo Alto, CA, USA, 1986.
[4] M. Auguston and A. Delgado. Iterative constructs in the visual data flow language. In *Proceedings. 1997 IEEE Symposium on Visual Languages (Cat. No.97TB100180)*, pages 152–159, Sep 1997.
[5] A. L. Davis and R. M. Keller. Data flow program graphs. *Computer*, 15(2):26–41, Feb 1982.
[6] N. Harvey and J. Morris. Nl: A parallel programming visual language. *Australian Computer Journal*, 28:2–12, 1996.
[7] D. D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages and Computing*, pages 69–101, 1992.
[8] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, Mar. 2004.
[9] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.
[10] P. R. Kosinski. A data flow language for operating systems programming. *SIGPLAN Not.*, 8(9):89–94, Jan. 1973.
[11] H. F. Levison. Swift. `https://www.boulder.swri.edu/~hal/swift.html`. Accessed: 2017-05-07.
[12] H. F. Levison. Swifter – an improved solar system integration software package. `http://www.boulder.swri.edu/swifter/`. Accessed: 2017-05-07.
[13] M. C. Lewis, H. Levison, and G. Kavanagh. Swiftvis: Data analysis and visualization for planetary science simulations. In *AAS/Division of Dynamical Astronomy Meeting# 38*, volume 38, 2007.
[14] K. Malawski et al. Reactive streams. `http://www.reactive-streams.org/`. Accessed: 2017-05-12.
[15] A. Prokopec and J. Suereth. Scalameter. `http://scalameter.github.io/`. Accessed: 2017-05-14.
[16] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins. Data-driven and demand-driven computer architecture. *ACM Comput. Surv.*, 14(1):93–143, Mar. 1982.
[17] P. C. Treleaven and I. G. Lima. Future computers: Logic, data flow, ..., control flow? *Computer*, 17(3):47–58, March 1984.
[18] P. G. Whiting and R. S. V. Pascoe. A history of data-flow languages. *IEEE Ann. Hist. Comput.*, 16(4):38–59, Dec. 1994.